universität
innsbruck

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.

Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

26.10.2020	Lorenz Schmidt
Datum	Unterschrift

Leopold-Franzens-Universität Innsbruck

Department of Computer Science
Security and Privacy Lab

Bachelor thesis

Visualizing the Databike

Lorenz R. Smidt

supervised by
Alexander Schlögl, MSc

Innsbruck, 27. Oktober 2020

Abstract

The Databike, a mountain bike equipped with a variety of sensors, can capture a range of properties while in use. This information is sufficient to reconstruct its state at any given point in time. A bike ride can be recorded in its entirety by recording these properties multiple times a second. In this thesis an application for interactively replaying a recorded bike ride is developed using the Unity engine. Additionally, the file format for a bike ride, called a timeline, is defined based on the GPX standard. The bike is visualized using an animated 3 D model of the Databike, which is purpose-made for this application, as well as various interface elements. The most important techniques used in the creation of the model are outlined as a part of this thesis. In order to allow for smooth and optionally slowed-down playback of the recorded data, the properties are interpolated using a cubic interpolation library. Combined with a video player-style interface, the user is able to skip through parts of the ride and review specific sections in detail. The finished application is available for both Windows and Linux thanks to the cross-platform build support of the Unity engine. Furthermore, the entire project is developed with expansion in mind: It contains provisions for adding more bikes, visualization elements, application settings, and different timeline file formats.

Zusammenfassung

Das Databike, ein Mountainbike welches mit einer Reihe von Sensoren ausgestattet ist, kann eine Vielzahl an Eigenschaften während dem Betrieb aufzeichnen. Diese Informationen reichen aus, um seinen Zustand zu einem bestimmten Zeitpunkt zu rekonstruieren. Dadurch kann eine Fahrt durch mehrmaliges Aufzeichnen dieser Eigenschaften pro Sekunde vollständig aufgenommen werden. In dieser Arbeit wird eine Anwendung zur interaktiven Wiedergabe einer aufgenommenen Fahrt mithilfe der Unity Engine entwickelt. Zusätzlich wird das Dateiformat für eine Fahrt, genannt eine „Timeline“, basierend auf dem GPX-Standard definiert. Das Databike wird mithilfe eines animierten 3D-Modells des Fahrrads visualisiert, welches speziell für diese Anwendung erstellt wird, sowie mit weiteren graphischen Oberflächen. Die wichtigsten Techniken zur Erstellung des Modells werden als Teil dieser Arbeit umrissen. Für eine flüssige und optional verlangsamte Wiedergabe einer aufgezeichneten Fahrt werden die Eigenschaften mithilfe einer Bibliothek für kubische Interpolation interpoliert. Kombiniert mit einer Videospieler-artigen Steuerung ist der Benutzer in der Lage bestimmte Teile der Fahrt zu überspringen oder im Detail zu betrachten. Das fertige Programm ist sowohl für Windows als auch Linux verfügbar dank der Multiplattformunterstützung der Unity Engine. Außerdem ist das gesamte Projekt für Erweiterbarkeit ausgelegt: Es enthält Provisionen um weitere Fahrräder, graphische Oberflächen, Programmeinstellungsmöglichkeiten und Dateiformate hinzuzufügen.

Contents

List of Figures

Listings

1 Introduction

The Databike has sensors mounted to it which capture its properties at any point in time. This information can be saved at regular intervals to record a sequence of bike motions. The data thus obtained is sufficient to accurately reconstruct entire bike rides.

In this thesis an application to visualize and review recorded bike rides is developed using the Unity engine. A number of parts are purpose made for this project including the bike ride file format, the custom 3 D model of the bike, and the user interface and controls of the application.

First a look at the most important techniques used for the creation in the 3 D model is taken in the sections 1.1, 1.2, and 1.3. Following that, pre-existing tools and methods which are used for the project are described in section 2.

With the groundwork complete, the project is developed in three distinct steps: First the 3 D model of the Databike is created in Blender, which is described in section 3.

The next step is parsing and processing the recorded data from the actual bike, which is detailed in section 4.

Finally, the processed data is visualized using a Unity application, created in section 5. The project architecture is based on a pre-existing framework providing core functionalities, which is described in section 5.1. This framework was previously developed by me to create easily expandable applications using the Unity engine. Once the background systems of the project (sections 5.4, 5.5, and 5.6) are set up, the visualization elements and user controls are created in section 5.7.

1.1 Introduction to 3 D Modeling

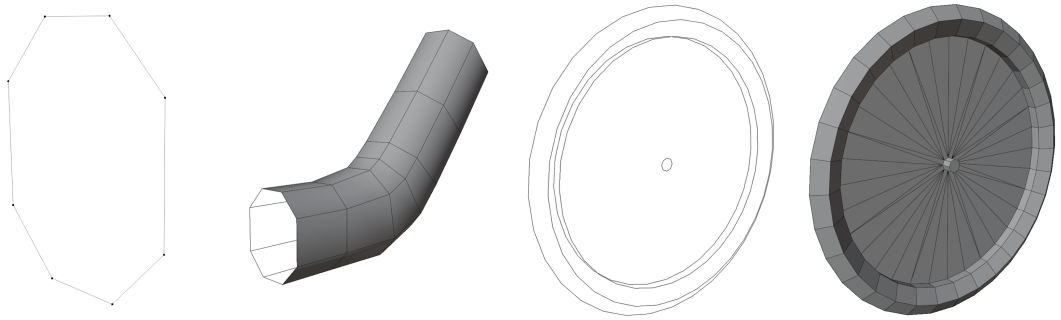
Three-dimensional models can be represented in a variety of ways, as detailed by Univ.-Prof. Dr. Harders in his lecture Computer Graphics [26]: Using implicit or parametric surfaces which are defined by mathematical functions, a collection of points which are dynamically interpolated, or using polygons. For the purpose of this project we will only focus on meshes defined as a collection of polygons.

A polygon mesh consists of a number of vertices, which are connected using edges. Two or more vertices can form a face or polygon, which has an orientation (i.e. front and back side). Connected faces sharing at least one edge form a mesh, which approximates a smooth surface using discrete points. The most common type of polygon used for meshes is a triangle with three vertices because it is inherently planar or flat. Depending on the tool used for creating and rendering meshes, any polygons consisting of more than three vertices are automatically converted into triangles, and most soft- and hardware is specialized to handle triangles efficiently.

All information in the following sections is based on the experience from creating the model of the Databike unless stated otherwise. All of the operations described are verified through experimentation and the sources given. The tools and operations described are based on Blender 2.81 [11].

1.1.1 Basic Geometry Creation

Reference Images Reference images are essential when accurately recreating a real-life object as a 3 D model. If the reference image correctly represents the dimensions of an object, meaning that it does not introduce (too much) distortion, it can be used as a direct size reference. In this case the mesh is modeled directly onto the image, transferring the dimensions to the model. Orthographic reference images, which introduce no distortions compared to the perspective projection, are perfect for this task. However, only two



(a) The primitive object.

(b) Result after repeatedly extruding.

(c) Features traced with edges.

(d) Mesh resulting from connecting traces.

Figure 1: Examples of the repeated extrusion and trace-and-translate methods.

dimensions can be reconstructed at a time using this method. The third one is created by either repeating the process with another reference showing the object from another angle, or by using a different modeling method.

Repeated Extrusion One of the most basic mesh creation techniques is repeated extrusion. Using it in combination with orthographic reference images allows for a fast and relatively accurate reconstruction of the object. First a template object is created which is then extruded multiple times following the shape given by the reference, creating steps in the mesh at locations of curvature or with more details. This is shown in figure 1a-b. Once the rough shape is established, the extruded pieces are combined into a single connected mesh. It serves as the basis onto which more details such as weld joints, pivot points, and other intricate features are added using other modeling techniques. The shape of the template depends on the object that is being recreated as well as further modifications made to the mesh. For example, the template for the central frame of the Databike model is a cube with beveled edges along one axis. This decision is detailed further in section 1.1.3.

Trace-and-Translate Another method for creating a 3 D model based on a single non-distorted reference image is to trace and translate: First the outline of the object is traced using edges, followed by the outlines of other visible features. This usually results in multiple disconnected strings or loops of edges. The features are then translated to roughly the right height / depth relative to the reference plane. This results in a rough lattice of edges describing the key dimensions of the object which are then filled in using faces. Figure 1c-d shows this process at the example of a clock face. The mesh thus created is further refined by eye to match the references more closely and additional details are created where needed. This method is most useful for smaller, solid objects for which only one non-distorted reference exists.

Additive Construction Additive construction is used for adding features with vastly different shapes to an already established mesh. The feature is created as a separate mesh which is moved into the correct relative position. The two meshes are combined into one by adding new edges at the locations where the two intersect. This is done using the `Knife (Self Intersect)` tool which computes the intersections between the selected faces and inserts new vertices and edges at the resulting points. With this new geometry in place

the two meshes are modified to both incorporate the newly created edges, resulting in one continuous mesh.

Subtractive Construction Instead of separately creating a feature and combining it with the rest of the structure, it can also be used as a stencil to remove part of the mesh. This method is called subtractive construction and is commonly used to create recesses in already established meshes. Similarly to additive construction, a usually more basic mesh in the shape of the recess is created and positioned at the correct location. After applying the `Knife (Self Intersect)` tool, all of the required edges are present in the mesh. However, in contrast to additive construction, only the part of the stencil describing the recess within the established mesh is incorporated. Any additional geometry from the stencil is discarded.

1.1.2 Adding Details to Geometry

Finer details are added to the mesh once the basic geometry is established using the methods described in the previous section. The primary method used for this is manual vertex manipulation. The following basic operations, as shown in figure 2, are used any number of times and in any order to achieve the desired result:

Translate (2): This basic mesh transformation moves the selection in a certain direction and can be applied to all kinds of geometry, from single vertices to multiple faces. It is primarily used to create changes in topography such as raising or lowering parts of the geometry, or to align parts of the mesh with references.

Rotate (3): Rotating part of the mesh is usually required to bring it into alignment with another part of the object. For example, the front suspension cylinders of the Databike model are rotated to match the axis of the steering column. A rotation is always relative to an axis of rotation, which is often one of the world coordinate axis.

Scale (4): Resizing or scaling part of the mesh can be used, beyond the obvious function of changing the size of the selection, as a radial translation tool: If all of the affected vertices are planar, scaling moves them farther apart or closer towards each other, similarly to multiple translation operations relative to the center of the selection. However, this operation does not preserve the length of affected edges in general. It is often used when working on meshes with radial symmetry.

Extrude (5): Extruding copies the selected geometry and inserts edges and faces between the new and old mesh, connecting the two. It works on any combination of geometry, from single vertices to entire meshes. After extruding a selection, a translation operation is automatically started, as the newly created geometry almost certainly needs to be moved to a different location than the old one. If this translation is not performed it can result in duplicate geometry or other anomalies, which have to be dealt with as described in section 1.1.4.

Subdivide (6): An edge is split in two using the subdivision operation, resulting in a new vertex being inserted in the middle. When applied to two edges of the same face at once, a new edge connecting the new vertices is created as well. The face in question is split in two as a result. This operation can be applied to any number of edges at the same time. For example, a strip of faces can be split lengthwise by applying the subdivision operation to all cross-wise edges. The `loop cut and slide` tool is an extension of this functionality, subdividing a mesh with radial symmetry around its circumference. However, in addition to inserting the new edges, it also translates the

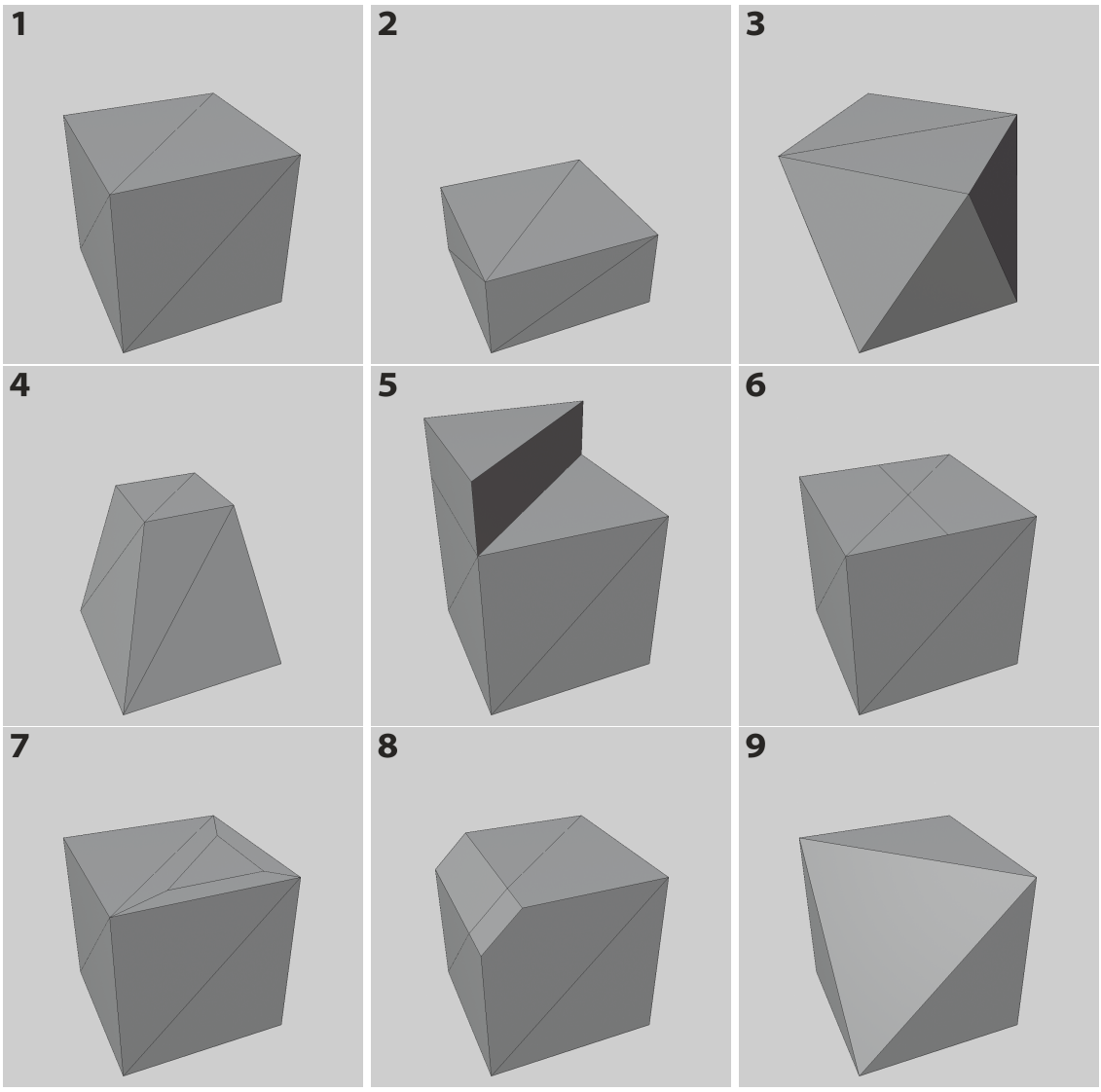


Figure 2: Overview of the basic operations applied to (part of) the top of a cube; (1) The base cube. (2) Translating the top faces downwards. (3) Rotating the top faces 45° clockwise. (4) Scaling the top faces to half size. (5) Extruding the left triangle upwards. (6) Subdividing the front, diagonal, and rear edges. (7) Insetting the right triangle. (8) Beveling the left edge. (9) Merging the front vertex with the rear left vertex at the rear position.

created edge loop along the original edges. This functionality is primarily used when working on cylindrical objects.

Inset (7): Similarly to the subdivision operation for edges, `inset` splits a face into smaller pieces. However, instead of splitting multiple edges at the center and connecting the new points, `inset` operates on a single face at a time. It creates a copy of all vertices defining the face and translates them towards its center, resulting in a new ring of faces around the perimeter. This operation is especially useful for reducing the diameter of a cylinder, or creating a raised or lowered section in the middle of a face in combination with a translation.

Bevel (8): The bevel operation replaces an edge with a face, resulting in a chamfer and

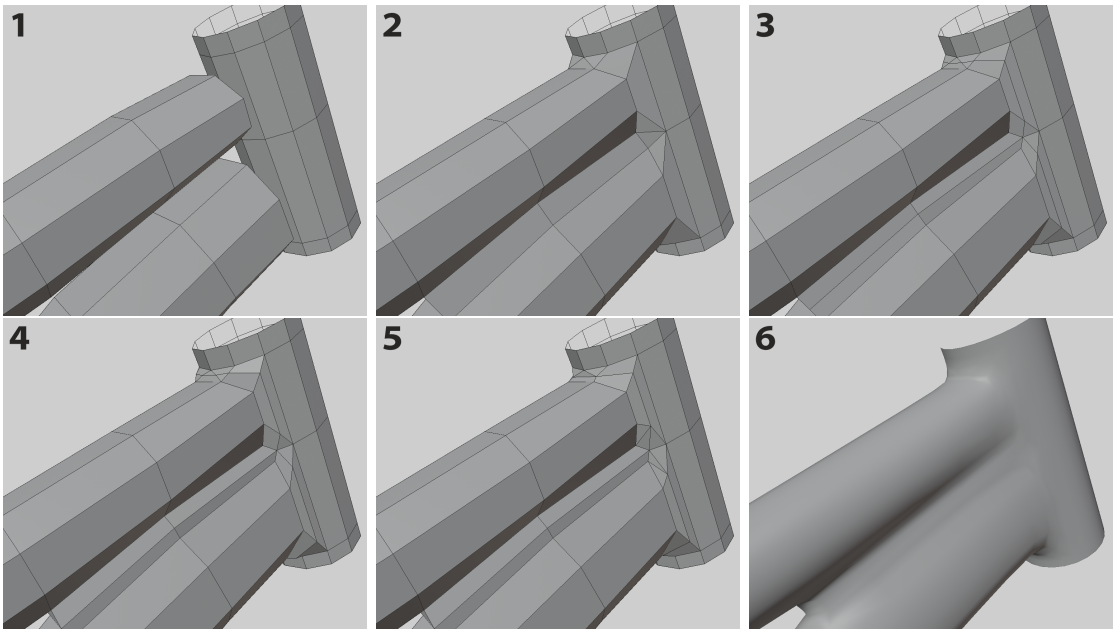


Figure 3: Example of details added to basic geometry, in this case the steering column of the central frame; (1) The basic geometry, created using repeated extrusion. (2) Connecting the multiple meshes. (3) Subdividing the connecting faces along their center line. (4) Translating the new edges outwards. (5) Modifying the T junction to shade correctly and merging redundant vertices. (6) The final result with the subdivision surface modifier applied.

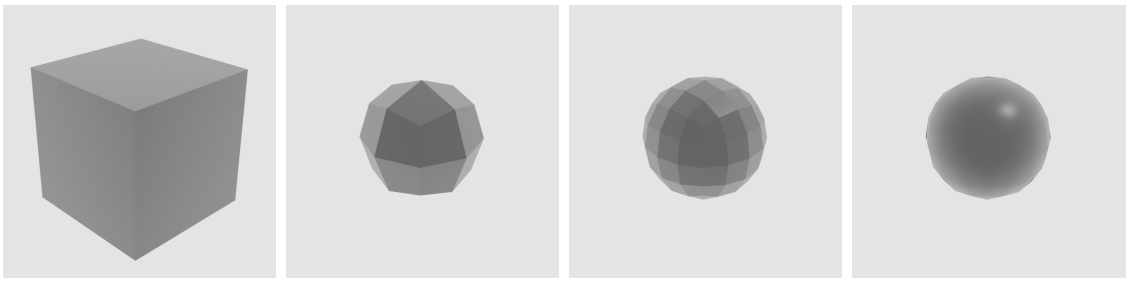
a less 'sharp' corner. The width of the new face as well as the number of faces to be used as a replacement for the edge can be specified. Using more faces results in an approximation of a rounded corner.

Merge (9): `Merge` combines two vertices into one, concatenating the edge between them if it exists. The resulting vertex can be created at either one of the original vertex locations, or at the center between the two. All edges and faces connected to the original vertices are adjusted to instead connect to the new one. This operation is primarily used to either remove redundant or unnecessary geometry from a mesh, or to connect two meshes together.

Most of these operations can be restricted to operate along a single axis. This functionality is frequently used with the translation, rotation, and scaling operations, as the default axis for both is the facing direction of the camera.

It should be noted that the above list is by no means exhaustive and represents only a small subset of all operations and tools used in the creation of the Databike model. They are however the ones most commonly used ones, and a lot of other functionality is based on them (e.g. translation along an edge, rotation around a specific point).

As an example, the following sequence of steps is used to add details to the steering column section of the central frame, as shown in figure 3: First the basic geometry is connected into a single mesh (pictures 1 and 2). The cross-edges at the seams are subdivided, creating a new edge along their center line (picture 3). The newly created partial edge loop between the upper and lower sections of the frame is scaled along the global x-axis, pushing it slightly further outwards from the center. The same is done to the partial loops around the joints between the frame sections and the steering column using multiple axis-restricted



(a) The base cube (b) One pass (c) Two passes (d) Two passes with smooth shading

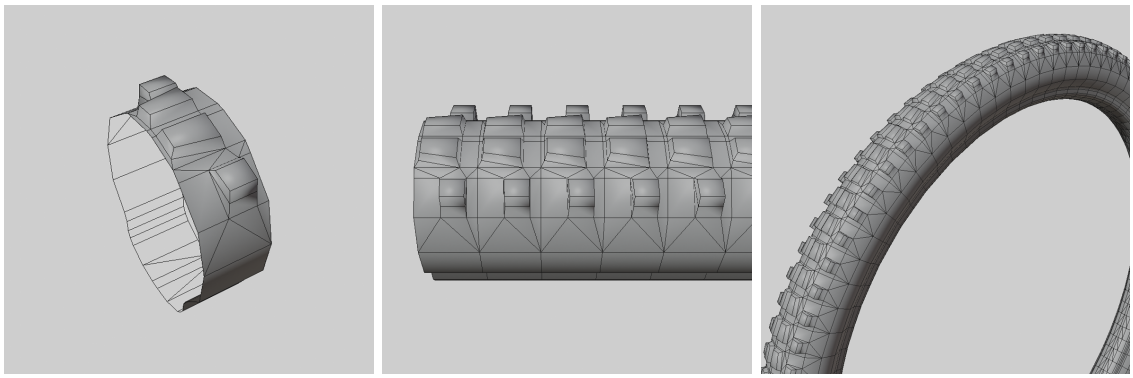
Figure 4: Example of the Subdivision Surface modifier with different numbers of passes being applied to a cube;

scaling operations (picture 4). At this point the result is checked with the subdivision surface modifier applied, which is described in detail in section 1.1.3. Most of the joints now have the desired raised appearance, offsetting them from the surrounding mesh. However, the mesh at the T junction shows unwanted sharp edges. This is resolved by removing the upper and lower quads and manually recreating them as triangles. Additionally, the upper and lower vertices at the junction are moved further apart using a scaling operation limited to the local y-axis (picture 5). Checking the visual appearance confirms that the desired result is achieved. As a final optimization of the mesh, the vertices at the top and bottom of the joints are merged into fewer. This does not impact the final appearance in any meaningful way, but still reduces the number of required vertices by a non-trivial amount, especially after applying the subdivision surface modifier (pictures 5 and 6).

1.1.3 Advanced Geometry Creation

Any object in Blender can have `Modifiers` applied to it which influence its properties and geometry in various ways. However, the basic geometry is not changed directly. The processed geometry is instead tied directly to the modifier, allowing the user to review the result before permanently applying it to the mesh. Furthermore, a modifier is always applied to the entire geometry contained in the object, meaning that excluding a section of the mesh from the operation is impossible. The section has to instead be separated into a different object which does not have the modifier applied. In this section the modifiers most commonly used in the creation of the Databike model are highlighted.

Subdivision Surface Due to the nature of vertex-based geometry, round objects are approximated using a number of straight sections. In order to increase the 'smoothness' of the resulting object, the complexity of the mesh needs to be increased. This approximation is automated using the `Subdivision Surface` modifier, as shown in figure 4. It takes an existing mesh and generates geometry of a higher resolution based on it, according to the Blender Manual [9]. This is done using the Catmull-Clark surface subdivision algorithm, the first version of which was presented by E. Catmull and J. Clark in 1978 [13]. According to the authors, it is based on a "recursive bicubic B-spline patch subdivision algorithm". The central and rear frames of the Databike, among other parts, receive this modifier, which is why the base mesh only acts as a guide for the subdivision algorithm. This further influences the decision to choose a primitive shape used for the repeated extrusion. Using a rectangle with beveled corners results in an oval cross-section, which matches the profile of the real-life Databike frame tubes. This shape was found using trial and error with the goal to use the minimum amount of geometry required. For the sharper features



(a) Base tire profile geometry (b) Replicated profiles using the `Array` modifier (c) Profiles deformed around a circular curve

Figure 5: Steps in the creation of a tire;

present in the frame, such as the pivot points for the rear frame, additional edge loops are inserted close to the sharp sections, resulting in more abrupt changes in the interpolated mesh.

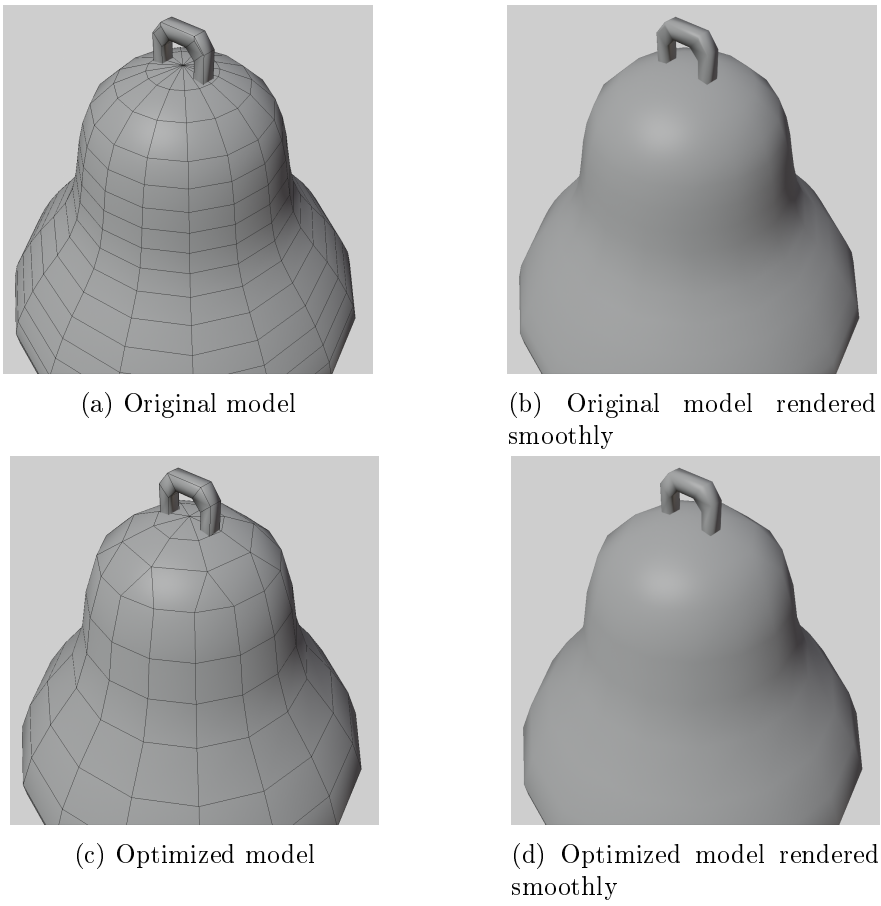
Mirror Another time-saving method is the use of the `Mirror` modifier. It replicates and mirrors the object along a specified axis, as stated by the Blender Manual [8]. This mirrored version is automatically updated with any changes to the original geometry, provided that the modifier is not permanently applied to the object.

Curves `Curves` are a special object type in Blender. According to the Blender Manual on the topic [3] they are defined as a set of control points which are interpolated, as opposed to the explicit vertices of meshes. They are available in both Bézier and NURBS formats, with the main difference between them being how they are calculated behind the scenes. In the case of the Databike model only Bézier curves are used. Bézier curves consist of control points with handles that define the curvature of a segment [5]. The curve is made up of multiple segments which are defined by two control points each. A curve is a closed loop if its first and last control points are the same.

Array and Curve Modifiers can be combined by applying them to an object in sequence. One especially useful combination is the `Array` with the `Curve` modifier. `Array` replicates the mesh of the object a configurable number of times along an axis, with each copy being offset by a specified amount [6]. `Curve` takes a mesh and deforms it to fit a given curve of either variety (Bézier or NURBS), as stated by the Blender Manual entry on the topic [7]. The result depends on the shape of the curve chosen for the deformation. Combining the two allows for a simple mesh to be repeated and transformed into a more complex shape. Figure 5 shows this process on the example of a tire from the wheels of the Databike.

1.1.4 Optimization and Cleanup

Redundant geometry is removed in order to reduce the model's complexity. Duplicate vertices at the same position influence the shading of the model as well. Combining them into a single vertex does not influence the model's appearance in a negative way (it tends to remove unwanted artifacts). For this task the `Merge by Distance` operation is used. It merges vertices that are less than a certain distance apart into a single vertex at their



(a) Original model

(b) Original model rendered smoothly

(c) Optimized model

(d) Optimized model rendered smoothly

Figure 6: Example of mesh optimization; The original contains 433 vertices and 854 triangles, the optimized one 313 vertices and 614 triangles.

averaged location. As with the normal `Merge` operation described in section 1.1.2, all edges incident to either of the merged vertices are incident to the resulting vertex.

Furthermore, any features of the mesh that do not significantly influence the final appearance are removed. Figure 6 shows this process on the example of a bell. Note how the removal of every second edge loop around the circumference does not significantly change the result. This is because there is only a minor difference between each one of the vertices and their neighbors. The goal is to reduce the mesh to its most important features and represent them by the least number of vertices possible.

Another source of unnecessary complexity is hidden geometry. Any faces that cannot be seen from the outside will always be discarded during the culling process, as described by Univ.-Prof. Dr. Harders in his lecture Computer Graphics [28]. However, they are still loaded into memory, increasing the rendering overhead. These faces, along with other invisible geometry such as vertices and edges that aren't part of any faces, are removed from the model. This type of 'loose' geometry tends to be created by accident during editing operations.

1.2 Introduction to Texturing

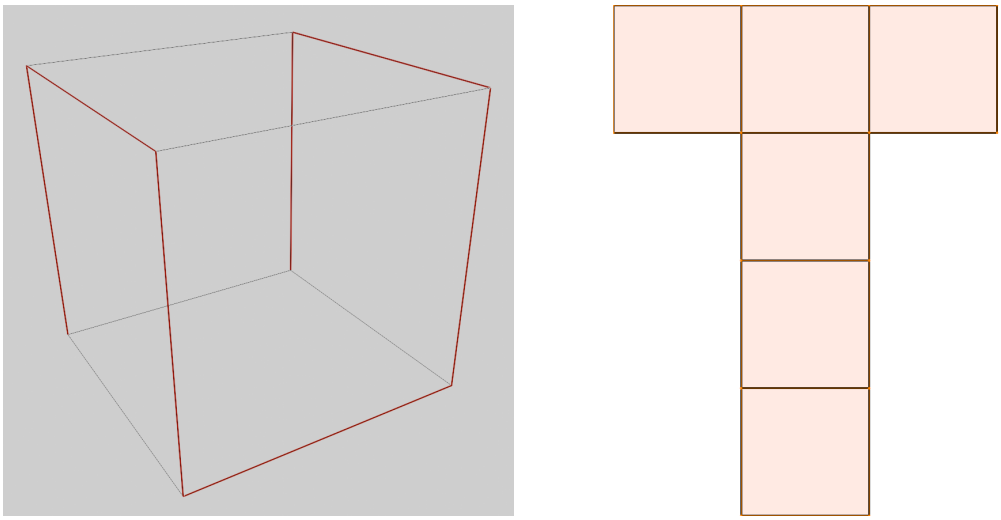
1.2.1 Textures

The visual appearance of a model is enhanced using textures, as described by Univ.-Prof. Dr. Harders [27]. The goal of applying textures to a model is to display images instead of

a solid color on the faces of the mesh.

The textures onto which the unwrapped mesh is projected are always square. As stated by Univ.-Prof. Dr. Harders [27], this is primarily because textures are automatically `wrapped` (not to be confused with (UV-)unwrapping) - duplicated side-by-side, so that any mesh outside the base size still receives a texture. However, when a picture is copied and the copy is simply placed next to itself (called `repeated` for texture wrapping) a seam is visible. In order to avoid having discontinuities like these in the final result, textures that are meant to be repeated are often `seamless`. This means that the left edge is the same as the right one (the same goes for top and bottom), resulting in smooth transitions when repeating them side-by-side. Creating textures with this property is time consuming, which is why a different wrapping method is available: `Mirroring`. If the copy of a texture is mirrored horizontally before being placed to the right of the original, the meeting edges are guaranteed to match. Using this method the texture can be duplicated without having any seams. The downside with all wrapping methods is that any distinct patterns in the texture (e.g. flaws or irregularities) are easily visible, especially if a large number of duplicates are visible at the same time (this can often be observed on terrain or grass textures in video games).

1.2.2 UV-Unwrapping



(a) The cube with the seams marked in red.

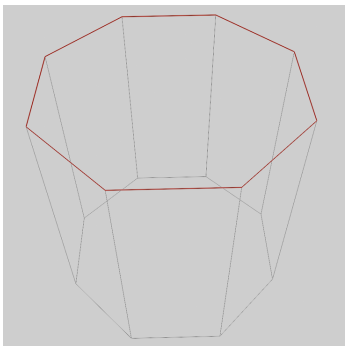
(b) The cube projected into 2 D space by cutting along the seams.

Figure 7: Unwrapping a cube by cutting along 7 edges.

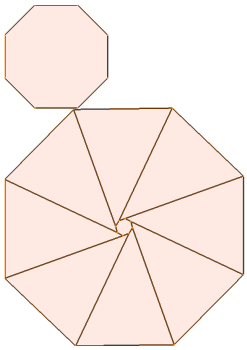
Because a model is three-dimensional, but a picture only has two dimensions, the 3 D mesh needs to be projected onto 2 D space. More precisely, the mesh needs to be `mapped` into texture space or `UV space`. The goal is to lay the mesh out flat, called `unwrapping` (or UV-unwrapping), without introducing distortions, as shown in figure 8. This is achieved by cutting the mesh along `seams`, which are likely to be visible in the end product as the textures do not meet up perfectly. Therefore the tactic is to use as few seams as possible and hide them in unimportant areas (e.g. the underside of the bike).

For example, a cube can be unwrapped by cutting along 7 edges, as shown in figure 7. These seams are shown in red on the left hand side. Seams have to be designated by hand, and choosing different edges results in different projections.

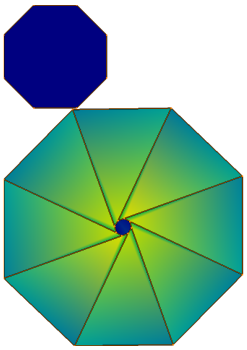
As noted by Univ.-Prof. Dr. Harders [27], the unwrapping procedure can be automated



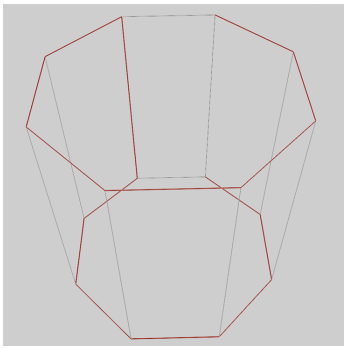
(a) Mesh with the seams marked in red



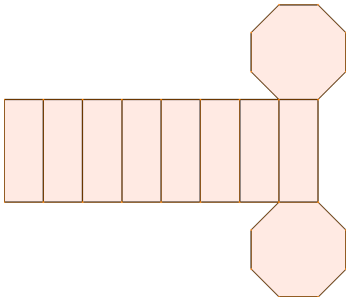
(b) Projected mesh



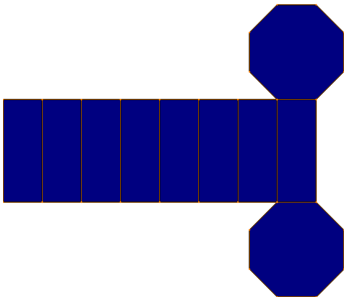
(c) Stretch analysis



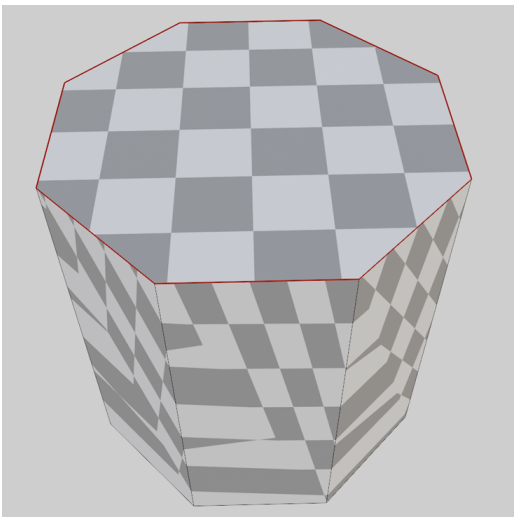
(d) Mesh with different seams



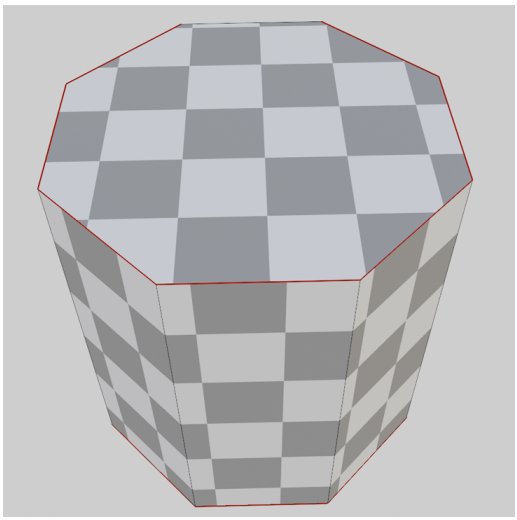
(e) Projected mesh



(f) Stretch analysis



(g) Checker texture applied to (a)



(h) Checker texture applied to (d)

Figure 8: Distortions introduced through non-optimal unwrapping; (c) and (f) show the stretch analysis with blue signifying no distortion;

for basic meshes such as spheres, cylinders and the like, with some methods capable of handling more complex shapes. However, for more intricate models manual unwrapping is necessary to achieve good results.

Distortions are the result of a mapping that fails to preserve the angles between a vertex' incident edges. An example of this is shown in figures 8a-c. The unwrapping process is automatically done by Blender using the `Unwrap` command. The strategy is to preserve as many angles of incidence as possible. By marking an edge as a seam Blender is allowed to use it multiple times when unwrapping, resulting in the faces being disconnected. This

can be observed in figures 8a-b, where all of the edges around the top face are marked as seams, resulting in it being completely disconnected in the mapping.

Based on this the optimal unwrapping would be to mark all edges as seams and simply map each face on its own into UV space. This would guarantee that no distortions are present when applying a texture. However, all of the edges would show discontinuities when textured, as each face is in its own place somewhere in texture space. This is where the second goal comes into play: Creating continuous projections.

Trying to preserve continuity across the entire mesh is (usually) impossible. Therefore a middle ground has to be found by placing the seams in 'acceptable' locations, usually out of sight or in less important parts of the mesh. In figures 8d-f a mapping preserving continuity around the circumference of the cylinder is shown. However, the top and bottom faces are still mostly disconnected from the sides, and one of the edges on the back has a visible texture seam. Still, it is a clear improvement from the mapping in a-c, as can be seen in 8g and h.

1.3 Armature-based Animation

Armatures are used to simulate how a restricted model moves, as described by the Blender Manual [1]. The most common application area for this method is humanoid characters. The main idea is to simulate the rigid underlying structure of a skeleton and how it restricts the movements of the body as a whole. This idea is directly translated into armatures, which are made up of individual bones. Connected bones always stay together and move as one, disconnected bones can move freely if not configured otherwise. The rotation of bones can be restricted to certain rotation axis and angles. Configuring all of these properties correctly allows the movement range of the model to be limited in exactly the right way to achieve the desired result (e.g. the human knee only rotates within a certain range of degrees around a single axis), as stated in the Blender Manual on the topic [2].

1.4 Selected Aspects of the Unity Engine

The Unity engine is primarily intended as a game engine and supports more than 25 different platforms. It provides tools to create both 2 D and 3 D applications and is heavily used in mobile and VR content creation, as described in an article by L. Metney [16] in 2017. However, its uses extend beyond the game industry to other sectors. For example, the car manufacturer Audi uses the engine to allow its engineers to visualize new designs in VR, as described by S. Edelstein [19] in 2018. It is also used for simulations and research, as shown by Yang et al. [31] in 2016. The engine is free to use for individuals and companies with an annual revenue of less than $100,000.

The fundamental objects in Unity are `GameObjects`. According to the Unity Manual [20], they are containers for `Components`, which in turn dictate the functionality of the objects. All `GameObjects` have a `Transform` (or `RectTransform` for overlay interfaces) `Component` attached to them, which represents their position and orientation within the scene. All objects are organized in a hierarchy with parent and child `GameObjects` [21].

The `prefab` system allows an arrangement of `GameObjects` to be saved as a reusable Asset [24]. These act as blueprints and can be replicated (or instantiated) to the scene at runtime. This allows for relatively easy dynamic creation of complex scene elements such as user interfaces (UIs) or 3 D models.

`MonoBehaviour` classes are the entry point for any custom code in Unity [23]. They need to be attached to `GameObjects` within the scene and receive certain engine event calls. These include `Awake` and `Start`, which are called in that order before the first frame is

rendered, `Update`, which is called once before the each frame is rendered, and `FixedUpdate`, which is tied to the frequency of the physics system.

2 Previous Work

For some parts of the project pre-existing solutions are used: Equations for converting GPS coordinates to ENU coordinates, a cubic spline interpolation library for C#, and the GPS XML schema.

2.1 GPS Coordinate Conversion

GPS coordinates are represented by a latitude (ϕ) and longitude (λ) in degrees and a height (h) in meters. These are relative to a datum or reference frame. In the case of GPS this is the World Geodetic System 1984 (WGS84). GPS coordinates describe a point on a geoid. However, many applications require coordinates with respect to a reference point and a local offset along fixed coordinate axis, also called navigation coordinates. One such conversion method was presented by S. P. Drake in 2002 [18]. He describes a method to convert any given GPS coordinates to offsets relative to a given point. These offsets are given along the axis pointing east, north, and up from the reference point, with the resulting coordinate system being called ENU.

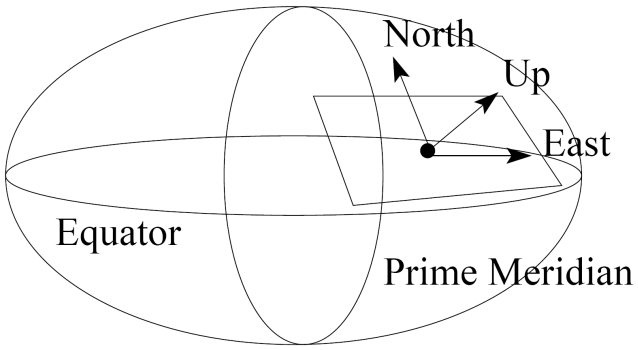


Figure 9: A local East-North-Up coordinate system shown in reference to the geoid [18].

The conversion process starts by translating the GPS coordinates into Earth Centred Earth Fixed (ECEF) coordinates. They describe the position on the geoid using three lengths along axis with the origin being the center of the earth. In this representation, points that are close together can be expressed as offsets along the axis relative to a reference point. These distances are computed from standard GPS coordinates ($\phi\lambda h$) using Taylor expansion about $\phi \to \phi + d\phi$, $\lambda \to \lambda + d\lambda$, and $h \to h + dh$, where ($\phi\lambda h$) are the GPS coordinates of the reference point and ($d\phi d\lambda dh$) are the relative GPS coordinates. The resulting displacements in ECEF coordinates are rotated to match the global north and east, and local up axis, as shown in figure 9. The terms of the Taylor expansion are truncated at the second order to increase the computation speed in exchange for an, according to the author, acceptable loss of precision. All of these steps are combined into three equations for the displacements along each axis, taken directly from [18]:

$$de = \left(\frac{a}{\chi} + h\right) \cos\phi \, d\lambda - \left(\frac{a(1-e^2)}{\chi^3} + h\right) \sin\phi \, d\phi \, d\lambda + \cos\phi \, d\lambda \, dh \tag{1}$$

$$dn = \left(\frac{a(1-e^2)}{\chi^3} + h\right) d\phi + \frac{3}{2} a \cos\phi \sin\phi e^2 \, d\phi^2 + dh \, d\phi + \frac{1}{2} \sin\phi \cos\phi \left(\frac{a}{\chi} + h\right) d\lambda^2 \tag{2}$$

$$du = dh - \frac{1}{2} a \left(1 - \frac{3}{2} e^2 \cos\phi + \frac{1}{2} e^2 + \frac{h}{a}\right) d\phi^2 - \frac{1}{2} \left(\frac{a \cos^2\phi}{\chi} - h \cos^2\phi\right) d\lambda^2 \tag{3}$$

de, dn, and du are the relative displacements along the east, north, and up axis respectively in ENU coordinates. ϕ, λ, and h are the GPS coordinates of the reference point, with $d\phi$, $d\lambda$, and dh being the difference between a given point and the reference in GPS coordinates. a is the semi-major axis of the earth, e^2 is the first numerical eccentricity of earth, and $\chi = \sqrt{1 - e^2 \sin^2\phi}$.

According to the author, these equations are accurate to within 10m inside 60km from the reference point.

2.2 Cubic Spline Interpolation Library

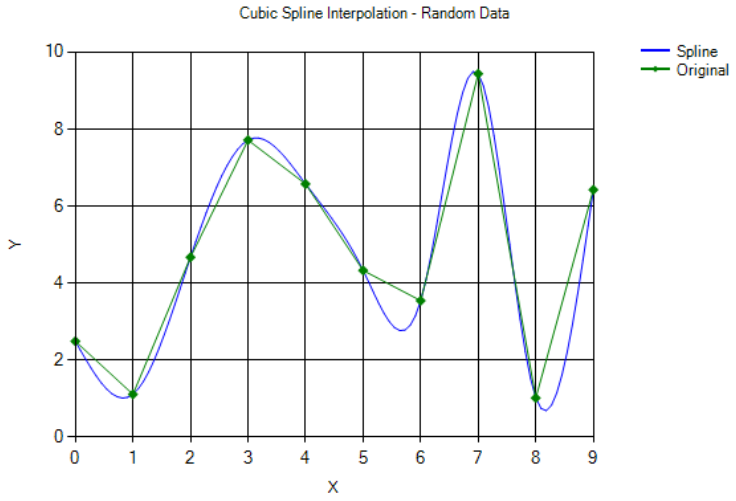


Figure 10: Example of spline interpolation from [17];

Spline interpolation has the goal of fitting connected curves over a set of data points, such that the values between the points can be computed. This can be achieved with a variety of equations, as described by the Wikipedia article on the topic [29] and by Seghers in his article detailing his implementation in C# [17]. In this case third degree polynomials are used, resulting in cubic splines.

In order to interpolate between N data points, $N-1$ splines q_i for $i = 1, 2, ..., n$ need to be constructed. The first and second derivatives q_i' and q_i'' of each spline need to match those of the previous (q_{i-1}) and next (q_{i+1}) ones at the meeting points to create a continuous transition. Additionally, the second derivates at the first and last data points are set to be 0, resulting in a 'Natural Spline'. These conditions are sufficient to solve the system of equations. This computation can be done in $O(N)$ time using the Thomas algorithm [30]. The resulting spline minimizes overall curvature, has continuous first and second derivatives, and contains all of the initial data points. However, it can produce

Figure 11: The final model with textures applied.

values that exceed the overall maximum or minimum of the initial points, as shown in figure 10 at X=8.

R. Seghers created an implementation of cubic spline interpolation in 2016 [17] in C# and made it open source under the CPOL license.

2.3 GPS Exchange Format

The GPS Exchange Format (GPX) is an XML schema designed for storing various GPS information. It is an open standard released in 2004 [12]. It provides definitions for disconnected waypoints, continuous routes describing a path, and tracks consisting of multiple track segments. The conceptual difference between routes and tracks are that the first are a suggested path and the second are recordings of already taken paths. All of these data structures consist of positions. Each position consists of at least the latitude, longitude, with potentially an elevation and a timestamp. The locations refer to WGS84 GPS coordinates.

All elements within the GPX format support optional extensions which allow for custom values to be stored in addition to the types provided. These are used by different manufacturers to store additional information such as street addresses, phone numbers, or temperatures, as shown by an extension schema by Garmin [14].

3 3D Model

The 3D recreation of the Databike is made in three parts: The mesh capturing the geometry, the textures which make the model look similar to its counterpart, and the armature and animations which capture its movements. The final result is shown in figure 11.

3.1 Creating the 3 D Model

References In order to recreate the real Databike as closely as possible a number of reference images are used. The most important one is the orthographic side view, which is shown on the website of Lapierre on the bike [15]. The image is used as a guide for all of the relative dimensions present in the bike. Using this in conjunction with the absolute measurements for the frame, as listed on the website, allows for accurate scaling of the model. In addition to the side view a variety of other reference images taken from the real Databike are used to recreate the more intricate details.

Central Frame The rough shape for the frame of the bike is created using repeated extrusion of a primitive shape, as described in section 1.1.1. Distinct parts are first created as separate pieces and joined together into a single mesh once their shape is established. For example, the upper and lower parts of the central frame are created separately and only joined together once they each match the reference images. The resulting mesh receives the `Subdivision Surface` modifier from section 1.1.3.

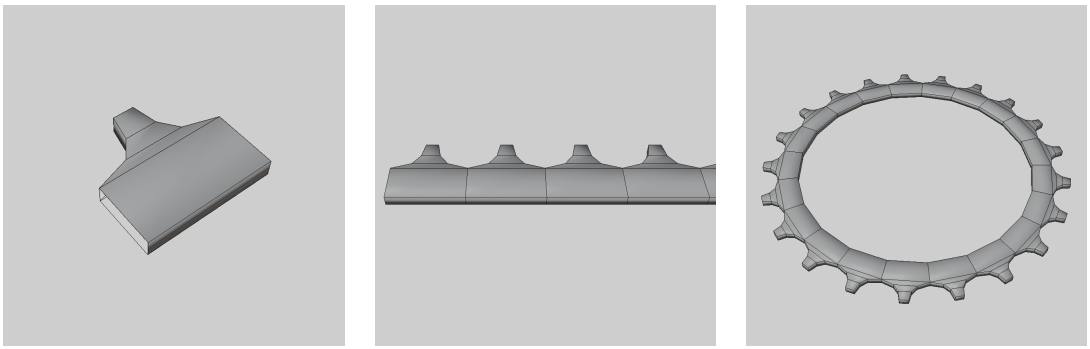
Frame Details The frame details, such as the pivot points or the attachment points for the rear suspension mechanism, are created in two steps: First the mesh is established using the trace-and-translate method described in section 1.1.1. It is refined by eye until it matches the references as closely as possible. It is then combined with the mesh of the frame using additive construction. The edges at which the two meshes meet are modified slightly after joining the two, improving the transition between them. This process is similar to the second half of the example described in section 1.1.2.

After the central frame is established, the steering column and front suspension fork are created. The fork as well as the front wheel hub are symmetrical along the central axis of the bike, which allows for the use of the `Mirror` modifier. Its functionality is described in section 1.1.3. Both of these parts have the modifier permanently applied to them after the mesh is established. This allows for side-specific features such as the mounting points for the front brake caliper to be added. The handlebar is created in a similar manner, as is the saddle, and some of the smaller parts like the pivoting link for the rear suspension mechanism.

The front suspension fork has a recess on top of each side. These are created using subtractive construction, which is described in section 1.1.1. The mesh used as a stencil is a primitive cylinder which is scaled and positioned to create the desired negative.

Small Parts All smaller parts of the Databike model are created using the trace-and-translate method described in section 1.1.1. The brake calipers, all objects attached to the handlebar, the pedals, and the chain guide assembly are created this way, among others. Reference images for all of these are taken from the real Databike beforehand.

Refinement Once the basic geometry of a part is established with the methods described above, it is further refined using the operations described in section 1.1.2. The part is complete once a reasonable level of detail is achieved and all important features are captured. This limit depends on the part in question. For example, the details of the steering column are less defined as the ones of the brakes attached to the handlebar. The reason for this is that the brakes have a more distinct shape, and in order for them to look realistic more details have to present. However, the steering column looks convincing enough with far fewer details in place.



(a) Base geometry for one tooth

(b) Part of the replicated teeth

(c) Array of teeth deformed into the shape of a circle

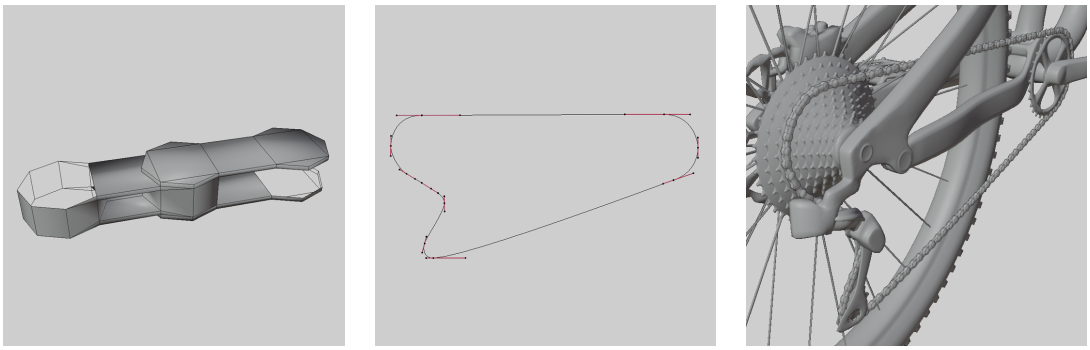
Figure 12: Steps in the creation of the gears;

Object Organization Independently moving parts of the Databike model are split into separate objects. This allows for easier rigging to the armature in section 3.3. For example, the central frame is separate from the steering column and the front suspension fork, which in turn are separate from the front wheel hub.

Gears and Wheels The combination of `Array` and `Curve` modifiers described in section 1.1.3 is used to generate the gears and wheels of the Databike model. In the case of the gears, the mesh to replicate is a single gear tooth, as shown in figure 12a. This tooth is replicated using the `Array` modifier to create the number of teeth required, resulting in a line as shown in figure 12b. Combining this with a Bézier curve modeling a circle using the `Curve` modifier results in the line being wrapped around the circumference. The curve is resized until the first and last teeth barely touch. After the shape is established both modifiers are permanently applied to the tooth object (in the right order), locking in the geometry of the gear, as shown in 12c. This process is repeated for each one of the gears present on the bike (the rear cassette consists of 10 gears with 11, 13, 15, 18, 21, 24, 28, 32, 37, and 42 teeth, the front gear has 30 teeth, and the idler 11).

The tires for the wheels are created in the same way, except using a section of the tire profile as the base mesh. Furthermore, instead of fixing the number of copies, the size of the circle is set to match the diameter of the wheels and the number of repetitions is increased until the circumference is filled. The circle still needs to be resized slightly until no overlap or gap is present between the first and last tire segments. This process is shown in figure 5.

Chain The chain of the bike is created similarly to the gears and tires. The base geometry for the `Array` modifier consists of one complete and one partial chain link. It is created such that the replicated versions fit together to form a complete chain, as shown in figure 13a. The offset between copies is changed in `Array` modifier to incorporate this partial overlap. Furthermore, instead of using a Bézier curve in the shape of a circle, a curve tracing out the path of the chain around the rear cassette, front gear, and tensioning mechanism is used (figure 13b). Because of the more complex curve, eliminating overlap or a gap between the first and last links is more difficult than with a circle: Adjusting the angle of the idler arm forwards or backwards lengthens or shortens the chain, respectively. This motion is used to establish the right length of curve through trial and error. The angle of the idler is animated in the final model, as described in section 3.3. This is because the motion of the rear suspension mechanism changes the distance between the cassette and the front gear.



(a) Base geometry consisting of one complete and one partial chain link

(b) Bézier curve tracing out the path of the chain

(c) Final result

Figure 13: Steps in the creation of the chain;

Wheel Spokes The wheels have 28 spokes each. They consist of seven groups of four which are spread around the circumference. This pattern is recreated using the `Mirror` operation, which works similarly to the modifier with the same name, but instead changes the geometry directly. Furthermore, it can be configured to mirror the mesh around a specific point. The mesh for each spoke is an elongated three-sided cylinder which connects one side of the central hub to the inside of the tire. Each one of the spokes is rotated by $(g \cdot 4 + i) \cdot a$ degrees around the center of the wheel. a denotes $\frac{360}{28}$, g the number of the group from 0 to 6, and i the position of the spoke in its group from 0 to 3. The first spoke in each group is simply rotated, the second one is mirrored along the x-axis, the third one is mirrored along the y-axis, and the fourth one along both axis. This process is shown in figure 14. The result is a similar four-spoke pattern as present on the real Databike.

3.2 Texturing

3.2.1 UV-Unwrapping

All meshes of the bike are uv-unwrapped according to the principles described in section 1.2.2. Some parts of the mesh only receive a basic black metal texture, so their mapping does not need to be as continuous as possible, as seams in the texture are less obvious. The parts which receive a custom texture, such as some sections of the central frame, are unwrapped in a way that all of the surfaces have virtually no distortions and no seams run through the special textures. This is shown in figure 15. Extra care is taken to place the seams at points of the mesh where two different textures meet (e.g. the grips on the handlebars). This results in distinct, already disconnected meshes once UV-unwrapped, making the positioning of the projection on top of the textures easier.

For some of the less critical or simpler parts an automated unwrapping tool is used. The `Smart UV Project` tool automatically places seams at all edges where the angle between the two meeting faces is greater than some specified amount. When applied to complex meshes the result is often either a lot of disconnected 'scraps', or a few highly distorted pieces. However, when the mesh is reasonably simple, such as the gears of the bike, the results are close to optimal mappings. The gears are essentially two flat and parallel faces whose outlines are connected. Applying the tool results in the two large faces being separated out and the connecting faces being laid out flat. This mapping fulfils all of the criteria stated in section 3.2.1. All of the gears as well as the brake calipers are unwrapped this way, with the reasoning for the latter ones being that their mappings are relatively unimportant.

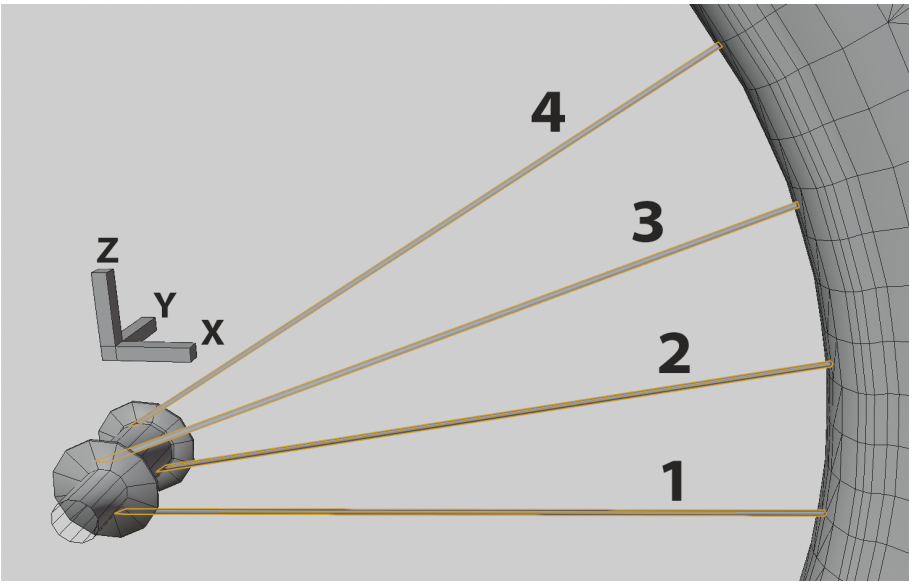


Figure 14: A group of four spokes, (1) only being rotated, (2) being mirrored along the y-axis before rotating, (3) being mirrored along the x-axis before rotating, and (4) being mirrored along both y- and x-axis before rotating; The coordinate system center is in the center of the wheel hub, not at the axis description.

3.2.2 Applying Textures

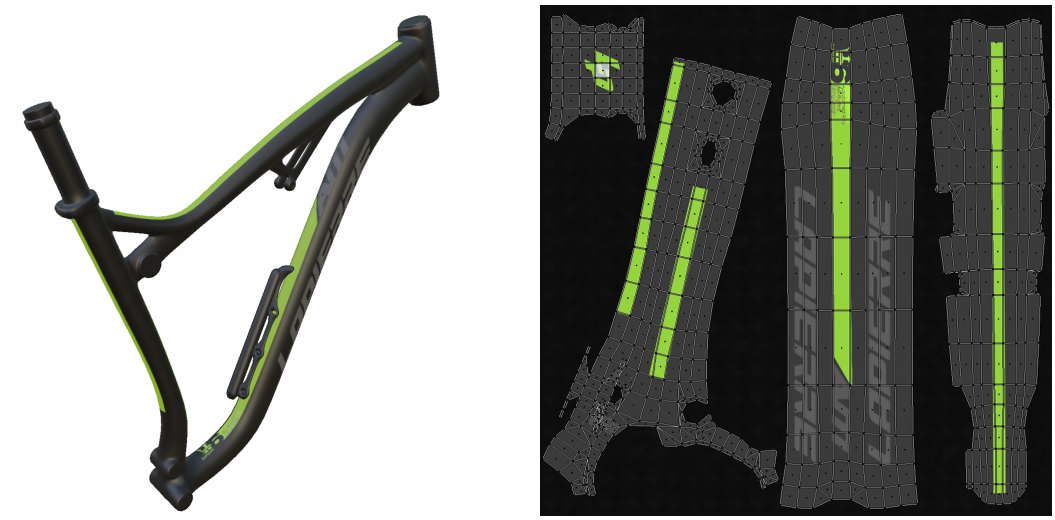
All of the textures used for the Databike model are based on reference images taken from the real bike. Textures are split into two categories: Basic and specialized. Basic textures capture simple structures such as black painted metal, bare silver metal, or rubber. They are used to color large parts of the bike at once and are reused across multiple objects. Basic textures are seamless and can therefore be easily wrapped, as described in section 1.2.1. Specialized textures capture details specific to certain parts of the bike. For example, the special texture for the central frame contains the distinct yellow-green stripes and the 'Lapierre' logo seen on the side, as shown in figure 15b. These textures are not wrapped. Because of their different application areas, basic textures are a quarter the size of specialized ones (512 versus 1024 pixels square), as they can make up for the difference through wrapping. The creation of textures from reference images as well as the methods used to make textures seamless are outside the scope of this thesis.

The finished textures are assigned to `materials`, which are then applied to the mesh. The UV-mapping dictates which part of the texture is shown on the geometry. Materials are configurable to have different shading properties, which are set inside the Unity editor in section 5.3.

3.3 Animating

3.3.1 Configuring Armature

Bone Structure Each major part of the Databike model is recreated as a bone with dimensions to match its critical features. In the case of the rear part of the frame, the bones extend between the pivots through which the parts are connected to each other. For rotating parts, such as the steering column or the pedals, the bones are positioned either along or orthogonal to the axis of rotation (both arrangements can be configured to capture the movement, the choice is one of intuition). The bones for the rear part of the frame are



(a) Central frame with textures (b) UV-mapping of the central frame

Figure 15: Example of UV-mapping for a specialized texture; Only faces receiving non-basic textures are shown.

arranged in a specific formation to allow the use of the `Inverse Kinematics` (IK) bone constraint, which is discussed later on in this section. Because the movement of the rear suspension mechanism changes the distance between the cassette and the front gear, the chain has to be animated as well. For this purpose a closed loop of bones tracing the chain is created. All bones are parented (directly or indirectly) to the bone representing the lower part of the central frame. The resulting armature is shown in figure 16.

Movement Constrains The bones of the central frame are restricted in all dimensions of translation and rotation, locking them in place. Bones representing rotating parts such as the steering column are restricted to only rotate around one axis. Parts sliding relative to their direct parents, such as the front suspension fork, have their bones restricted to no rotation and only one direction of translation. In the case of the Databike model, the maximum extend of each one of these motions, either rotation or translation, is not directly limited by bone constraints. Instead, they are indirectly set when creating the animations for these motions.

Cylinder Tracking The cylinder for the rear suspension mechanism consists of the cylinder body and the piston. These parts are represented by two different bones, one connected to the central frame, and one to the swinging link. In order to maintain the alignment of these bones towards each other, the `Track To` bone constraint is added to both of them. Selecting the corresponding other bone as the target causes them to automatically rotate to point towards the base of the other, maintaining the desired alignment.

Rear Frame Suspension Inverse Kinematics When the rear wheel is pushed upwards and the suspension cylinder compressed, the wheel pivots on the lower part of the rear frame. This motion is shown in figure 17. In order to capture this motion in the armature, the `Inverse Kinematics` (IK) bone constraint [4] is applied to the lower rear frame bone. It ensures that the end of the bone stays in place at the pivot point above the pedals, causing it to rotate relative to the upper part of the rear frame. This motion spreads the lower and upper halves slightly apart during the suspension action. The IK constraint is configured to only allow rotation around one axis, restricting the motion to the desired

Figure 16: The armature overlaid on top of the geometry. The bones colored in yellow are subject to the `Inverse Kinematics` constraint.

direction. During the animation process the piston of the rear suspension cylinder is moved towards the cylinder housing, compressing the cylinder and causing the pivoting link to swing forward. This motion is translated to the upper part of the rear frame, causing it to move forward as well. At this point the IK constraint on the lower frame causes the upper frame to rotate, ensuring that the distance between the front and rear pivot points of the lower frame stays the same. The resulting motion causes the rear wheel hub to move up- and forwards. In the real Databike this motion works the other way around, with the rear wheel travelling upwards causing the system to pivot around the lower part of the frame and compressing the cylinder. However, the method described for the armature results in the same motion, but allows for a more linear motion to be captured, as the vertical position of the rear wheel is not directly proportional to the travel of the suspension cylinder.

Chain Inverse Kinematics Due to the slight rotation between the upper and lower parts of the rear frame during the suspension motion, the distance between the front gear and the rear cassette varies. In order to adjust the chain according to this motion it is modeled with a loop of bones anchored at the upper part of the tensioning mechanism. The lower and upper parts of the chain receive IK constraints, causing them to stay connected to the front gear and the top of the cassette respectively. When the rear suspension moves, the top part of the chain causes the chain around the front gear to rotate counter-clockwise, pulling the lower part forward. This motion is then counteracted by the tensioning mechanism moving forwards, resulting in the chain staying connected.

Once all bones are configured correctly, the mesh of the model is assigned to its respective bones, causing it to stay rigidly connected to them when they travel. This is shown on the example of the rear suspension in figure 17.

3.3.2 Creating Animations

The animations are created using keyframes. In this case a keyframe captures the properties of (a part of) the armature and saves it at a position on the timeline. If multiple keyframes are present on a timeline, the properties are interpolated between them, resulting in a

Figure 17: Motion of the rear suspension mechanism, with and without bones shown.

continuous motion. This functionality is described in detail by the Blender Manual [10].

Each one of the motions that the Databike model can perform is created as a separate animation. For each one a keyframe capturing the at-rest position of all bones involved in the motion is inserted at the beginning. At the end of the animation a keyframe for the maximum travel of the motion is added. The transition between the two states is automatically interpolated. The interpolation type used is set to linear in order to ensure that the progression through the motion is directly proportional to the elapsed time in the animation. For some animations, such as the handlebar rotation, an additional keyframe is added in the middle of the animation, ensuring that the halfway point of the motion is where it is supposed to be.

4 Bike Data Processing

4.1 Recorded Data and Format

During a ride with the Databike the values describing the properties of the bike are recorded. In order to achieve good reconstruction accuracy, multiple snapshots are taken every second. Each one contains a complete description of the bike using the following data points:

Location and Time as WGS84 GPS coordinates and an ISO 8601 timestamp string, respectively.

Rotation of the bike in space. It is split into rotations around three axis: x towards the front of the bike, y towards the right side, and z being up (0 pointing towards north). Furthermore, there are two different types of rotation values: absolute and additive. Absolute rotation is from within the interval [0..360) and is only required at the start of the timeline. Additive rotation is has to be present in every timestamp. These rotations are from the interval $(-360..360)$ and are added to the previous orientation of the bike.

Wheel Rotation in revolutions per minute.

Steering Rotation in degrees between $-180°$ and $+180°$ for rotations to the left or right.

Pedal Rotation and Rotation Direction. The rotation is absolute ([0..360)) and combined with the current rotation direction (clockwise or counter-clockwise) to reconstruct the accumulative rotation. This is done to ensure that even if the pedals are turning very fast their rotation is still captured with reasonable accuracy.

Front and Rear Gears as the selected gear number. In the case of the Databike there exists only one front gear, but the system supports multiple ones as well.

Right and Left Brake Lever Positions as a fraction of their maximum travel, 0 being at rest and 1 being fully activated.

Front and Rear Suspension Travel in millimetres of cylinder travel with 0 being fully extended.

Seat Position in millimetres with 0 being in the uppermost position.

The data is stored as a GPX file using the extensions provided by the format, as described in section 2.3. Additionally, the timeline information such as the recording name and the bike used, are stored using GPX metadata extensions.

All of the bike's static properties are specified in a bike definition file. It is an XML file containing the following information:

- Bike Name as a unique identifier.
- Prefab Path for loading the correct 3 D model for the bike.
- Number of Front and Rear Gears.
- Maximum Front and Rear Suspension Travels in millimeters.
- Maximum Seat Travel in millimeters.
- Brake Lever for the Front Brake, either `left` or `right`.

4.2 Readout and Conversion

The recorded data is read from the GPX file by the `XMLTimelineReader` and parsed into a `RawTimeline`. The readout is done using an `XMLReader` and a state machine that steps through the file, extracting the desired information. As soon as the bike name is read, the corresponding bike definition file is loaded. It is used to convert some of the read values into a bike-independent format during readout, as well as for various other things during the playback phase. The individual timestamps are read in an implicit loop controlled by the state machine. The values are temporarily stored in a struct until the entire timestamp is read. They are then checked for completeness, converted or transformed if necessary, and added to the `RawTimeline`. All of the methods for the conversions are contained within the `UnitCoverter` class.

The recorded rotations $(x_r y_r z_r)$ of the bike refer to the axis described in section 4.1. They are aligned with the axis used by the 3 D model within the scene using the formula $(x_s y_s z_s) = (y_r z_r x_r)$. Additionally, if the rotation is an absolute rotation, y_s is incremented by 180° to align the zero-point with the in-scene north direction (= positive z-axis).

The absolute spatial rotations of the bike are in angles modulo 360. However, bike rotations need to be stored as global absolute rotations in order to allow for smooth interpolation between them. For example, if $x_s = 15°$ and it is changed by $-20°$ in the next timestamp, $x_s = -5°$, and not 355°. The following algorithm converts a modulo absolute rotation (modAbsRot) to a global absoulte one (globAbsRot) based on the previous angle (prevRot) and the assumption that the difference between the previous and current angle is less than 180°:

Listing 1: Converting a modulo rotation to a global absolute rotation

```
float ConvertSpatialRotation(float modAbsRot, float prevRot) {
    int div = (int)(prevRot / 360);
    float rem = prevRot % 360f;
    if (rem < 0) rem += 360f;

    float rel = (prevRot >= 0) ? modAbsRot : (modAbsRot - 360f);

    if (Mathf.Abs(rem - modAbsRot) <= 180f)
        return div * 360f + rel;
    else if (rem > modAbsRot)
        return (div + 1) * 360f + rel;
    else
        return (div - 1) * 360f + rel;
}
```

Similarly, the pedal rotation, which is given as a modulo absolute angle and a rotation direction, is converted to a global absolute angle. The main difference is that, because of the given rotation direction, the assumption that the difference is less than 180° instead becomes that it is less than 360° in the given direction.

The GPS location of the bike is converted into ENU coordinates using the equations detailed in section 2.1. The result is transformed such that the in-scene x-axis corresponds to east, the z-axis to north, and the y-axis to up.

The resulting `RawTimeline` consists of a list of values for each one of the properties. The elements at index i from each list represent the i-th timestamp of the timeline. This structure is chosen to make the conversion in the next section easier.

4.3 Interpolation

Once the timeline is read from the file and parsed to a `RawTimeline`, it is passed to the `TimelineConverter` service. The conversion into an `InterpolatedTimeline` starts by

computing the relative time offsets. The time for each timestamp in the `RawTimeline` is given as a Windows file time. This is converted into a floating point offset from the start of the timeline in seconds by computing the difference between each one and the start, and dividing it by 10^7. The resulting list of time offsets is used to interpolate the properties of the bike.

The interpolation is done using the C# cubic spline interpolation library by R. Seghers [17], as described in section 2.2. Each one of the lists of values for the properties is passed to a separate `CubicSpline` in conjunction with the time offsets list. The result is a `CubicSpline` instance which is set up to be sampled from at any time between the start and end times of the timeline.

These instances are combined within the `InterpolatedTimeline` data structure, which acts as the input for the `SampleSystem`. It also retains all of the supporting data of the timeline such as its name, the total length, and the file-specified pedal offset.

4.4 Sampling

The `SampleSystem` is responsible for sampling from the `InterpolatedTimeline`. It is set up to compute a fixed number of samples at the same time and provide them in the correct order to the `PlaybackSystem`, which is described in section 5.4. The samples are stored as `BikeState` instances, which contain a value for each one of the bike properties and act as simple data containers. The `SampleSystem` reuses all `BikeStates` by storing still to-be-delivered samples in a list and moving them to another after sending a reference off to the `PlaybackSystem`. New samples are generated when no more pre-computed ones are available. The 'old' `BikeState` instances are refitted with the new values and placed back into the to-be-delivered list. This is done to keep the memory footprint of the system relatively low by reusing the data structure instances.

The `CubicSpline` class supports sampling of the function at multiple points within the same call. This is more efficient since the values to sample are in ascending order, allowing for easier selection of the appropriate spline sections [17]. The sampling process starts by computing an array of time offsets, which are each *timestep* seconds apart. This array is then passed as input to the `Evaluate` function of the `CubicSpline` instances for the different properties. Because cubic spline interpolation can result in values exceeding the maximum or minimum of the inputs, the results are clamped to their respective max and min values where applicable. The current speed is computed from the location data using the difference between each point and the next. The resulting values are combined into individual `BikeStates`, each one describing the bike at a specific point in time.

The *timestep* size is computed based on two factors: The physics engine update tick time (*ptt* in seconds), and the current playback speed. During playback the scene is updated with a new `BikeState` once every physics update tick. The physics update cycle is used instead of the frame update cycle because the time between these ticks (*ptt*) is guaranteed to be constant, whereas frame render times can vary. Therefore, if the computed samples lie exactly *ptt* apart, the timeline is played back at real time speed. In order to slow it down, the time between the samples is reduced to ptt/x where x is the slowdown factor. For example, half-speed playback requires a *timestep* size of $ptt/2$. By default, *ptt* is 0.02 seconds, but can be set to any value in the Time settings of the Unity project within the editor [25].

Overall, the `SampleSystem` is built to be close to memory-neutral. This means that it reuses close to all data structures and memory needed during the sample computation, as well as for the samples (= `BikeStates`) themselves. Because the system is constantly active in the background during playback, it would otherwise create potentially large amounts of work for the garbage collector. This could in turn lead to stuttering during playback.

The samples are retrieved from the `SampleSystem` via a single function. This way the entire caching system is transparent to the rest of the project. In order to do this the system keeps track of the current playback time. Whenever the current time or speed is changed all cached samples are invalidated, causing the system to re-compute them with the new values the next time a sample is retrieved.

5 Unity Project

The Unity project is created in several parts, starting with the backend systems and ending with the user interface (UI). However, all of this is based on a pre-existing framework, the `SmidtFramework`. Since a number of its elements are an integral part of the functionality of the project, the framework is described in detail in the next section.

5.1 Excursion: SmidtFramework

The `SmidtFramework` provides functionality for UI and object management, a logging system and console, as well as cover functions for the Unity engine update cycles and input system. It was developed by me in the summer of 2019 as a way to create easily expandable applications using Unity, and was updated and adapted slightly to fit this project.

The framework has three main goals:

- Disconnect `MonoBehaviour`-specific functionality from the need to have the scripts attached to active `GameObjects` in the scene
- Provide a uniform keyboard input system
- Add a console for executing custom commands at runtime and reading log messages in a standalone build

It achieves these goals using the following four systems:

5.1.1 ControllerSystem

The `ControllerSystem` provides a way to dynamically manage objects in the scene during runtime. Objects are tied to controllers, of which there are two types: `BaseControllers`, which are for 3 D world objects, and `BaseUIControllers`, intended for overlay UI elements. It is also possible to have controllers in a management-only role without a corresponding visual object (see section 5.5 for an example). All controllers are Singletons and managed by the `ControllerMaster`. It handles controller creation, removal, retrieval, and showing / hiding of UI. This functionality is globally accessible from any class. Whenever something is done to a controller through the manager, an event function is called to update the controller's state: On creation, deletion, showing (= enabling), and hiding (= disabling), with the last two only being available for `BaseUIControllers`.

Without this system the `MonoBehaviours` need to be attached to the scene objects they are managing as `Components`. If the visual representation is swapped out for a different prefab all other application components accessing the `MonoBehaviours` must do so through the new object, and therefore be modified as well. Furthermore, nothing guarantees that only one instance of the object is active at a time. Finally, speaking from experience, UI management without a centralized access point, as provided by the `ControllerMaster`, has a tendency to quickly become confusing and messy.

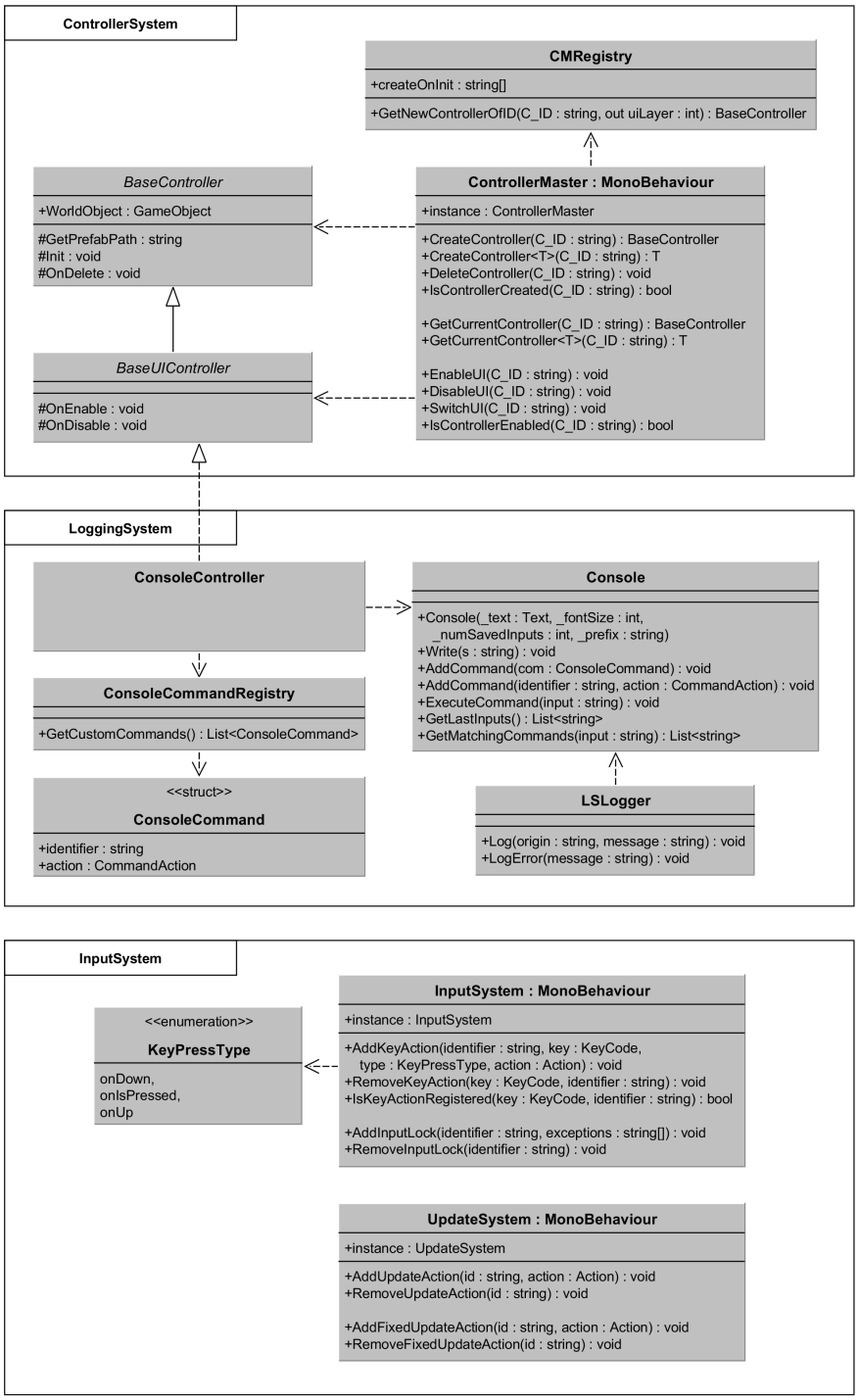


Figure 18: Class diagram of the `SmidtFramework`;

This system allows the code for each scene or UI element to be completely separated from its scene object, with all general management being handled by one central point. Controllers can obtain references to each other through the manager, allowing them to interact with each other without having a direct connection. Furthermore, showing or hiding UI elements is handled by a single component of the system, reducing the potential for interface flows not working correctly.

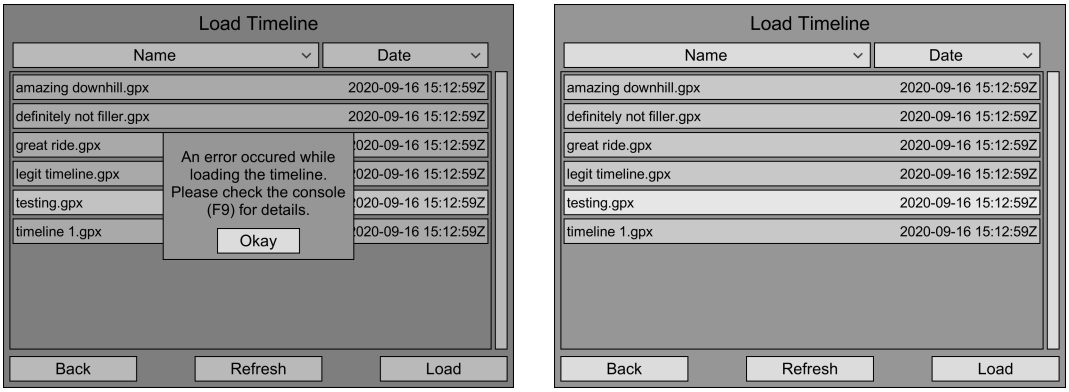
The `ControllerMaster` is a `MonoBehaviour`, allowing it to use the Unity `Awake` and

`Start` events, and functions as a Singleton. Internally, the currently active controllers are tracked using a hash map, or `Dictionary` in C#. This allows for efficient management of a large number of active controllers at the same time.

5.1.2 InputSystem

A keyboard button input can be processed in Unity by calling a method to check if the key is currently pressed from within the `Update` method. However, this functionality is only available for `MonoBehaviours`, since only those receive the required event calls from the engine. The `InputSystem` provides easier as well as unified access to this functionality to any class in the project.

The main feature of the system is the ability to register delegate voids (= `System.Action`) to be executed when a certain key is pressed. Additionally, the input type can be set to either when the button is first pressed down, released, or while being held. All keyboard inputs are registered with a unique ID. This is used to remove them from the system once they are no longer needed. Multiple inputs can be registered for the same key as long as their IDs are different.



(a) A dialogue being shown on top of another window; The dialogue has an input lock registered to prevent the underlying window from processing keyboard inputs;

(b) When pressing escape the dialogue window is closed and the Timeline Menu is able to process keyboard inputs again;

Figure 19: `InputSystem` input locks for stacked windows; Both windows have keyboard actions for the escape key registered, but only one is able to process inputs at a time;

In addition to this the `InputSystem` provides the ability to create so-called input locks. These are used to prevent all but certain keyboard inputs from being activated. When a new input lock is registered a list of keyboard action IDs is specified as exceptions to this lock. The most recent lock is always the one currently active, resulting in a stack-like hierarchy. Figure 19 shows this behaviour with an example.

The Unity engine does not provide any functionality similar to input locks by default. If a keyboard input is processed within the `Update` function of a `MonoBehaviour`, it will always be triggered if the key is pressed. Furthermore, multiple different `MonoBehaviours` can handle keyboard inputs at the same time, making it difficult to prevent all but certain actions from being executed. The `InputSystem` solves this problems by providing a unified service for managing keyboard inputs. Because the system has a complete view of all input actions at any time, the implementation of input locks is possible.

Similarly to the `ControllerMaster`, the `InputSystem` is a Singleton `MonoBehaviour`. Internally, additions and removals of a keyboard actions are buffered until after the current update cycle using the `LateUpdate` unity method. This allows input actions to un-register

themselves or others from the system while maintaining deterministic behaviour. Otherwise, the execution order of input actions would determine which get executed and which do not.

5.1.3 UpdateSystem

The `UpdateSystem` provides general-purpose access to the unity `Update` and `FixedUpdate` methods. Similarly to the `InputSystem`, it allows registering of delegate voids (= `System.Action`) to be executed, except that they are invoked regardless of keyboard input within the desired update cycle method. This allows classes to execute code once every frame or physics update while not being a `MonoBehaviour` themselves.

As with the previous systems, the `UpdateSystem` is a Singleton `MonoBehaviour`. Additions and removals of actions are buffered until the end of the respective cycle for the same reasons as with the `InputSystem` buffering.

5.1.4 LoggingSystem

The Unity engine provides an in-editor console by default. It is globally accessible and allows neutral as well as error messages to be written. Additionally, it allows for jumping to the position of the code which created the entry. However, no commands can be executed through the console at runtime, as it is essentially an interactive log file. Furthermore, it is only available from within the Unity editor and not the finished application. The `LoggingSystem` with the `LSLogger`, `ConsoleController`, and its console UI provide this missing functionality within the framework. Table 1 shows a comparison between the two systems.

Functionality	Unity console	LoggingSystem
Neutral messages	yes	yes
Error messages	yes	yes
Jump to source	yes	no
Available in standalone	no	yes
Execute commands	no	yes

Table 1: Comparing the Unity console to the LoggingSystem.

The `LSLogger`, which stands for `LoggingSystemLogger`, provides a globally accessible method to log messages to the console. Each message has a source provided with it, which is prepended in the console output to allow for easier reading of entries. All messages are mirrored to the Unity built-in console by default to make use of the jump-to-source feature it provides. Errors are always mirrored to the Unity console and are highlighted in red within the console UI.

The UI for the console is managed by the `ConsoleController`, which in turn is a `BaseUIController`. It allows previous inputs to be recalled as well as autocompletion of partially typed commands. The keyboard inputs are handled through the `InputSystem` in combination with the event functions provided by the `BaseUIController` class.

The console input is forwarded to the `Console` class, which is responsible for managing and executing commands in the form of `ConsoleCommand` structs. `ConsoleCommands` consist of the following two attributes:

- An identifier, the first continuous string to be input,
- and a `CommandAction`, a delegate void taking a list of strings as an argument.

The argument of the `CommandAction` is all parts of the input after the initial command split into substrings based on spaces.

`ConsoleCommands` are added to the system by means of the `ConsoleCommandRegistry`. The list of commands provided by the class are registered with the `Console` at the start of the application.

5.2 Project Architecture Overview

The Unity project is organized into two general layers: Controllers for visual and interactive elements, and services for independent computations. The central point of the project is the `SceneController`, which is created at application start using the `ControllerSystem`'s create-on-init functionality. It is described in detail in section 5.5.

All visual elements are tied to controllers of either the type `BaseController` for objects in the world, or of the type `BaseUIController` for overlay UI elements. The controllers are sized to only include one area of responsibility each. For example, the animation of the 3 D model is separate from the trail describing the bike's path in space, and they each have their respective controllers. All of the visual controllers are described in detail in section 5.7.

Computations independent of the visual representation are done within services. These are independent classes performing specific tasks which get invoked by controllers.

5.3 Setting up the 3 D Model

The 3 D model is exported from Blender into Unity as an fbx file. The materials for the textures are configured to create a realistic lighting behaviour such as specular highlights for the different kinds of metals.

The model's initial orientation is established by placing it in the scene and positioning it upright and facing north (towards the positive z-axis). It is then saved as a prefab and dynamically instantiated at runtime. This allows the bike model to be switched to other bikes depending on the loaded timeline.

The animations of the bike are imported along with the model itself. An `Animator` is set up to contain an animation layer for each of the animations. The layers are set to be additive, allowing multiple animations to influence the model at the same time.

5.4 Scene Playback and Update

Playback of a timeline is managed by the `PlaybackSystem`. It provides functions for pausing, playing, and changing the replay speed. It is the only class that interacts directly with the `SampleSystem`, which is described in section 4.4.

The system retrieves and passes samples to the `SceneController` while the timeline is being played. This is done once every physics engine update tick (or `FixedUpdate` tick) by registering a function with the `UpdateSystem`, which is described in section 5.1.3. This function is removed when playback is paused and re-created when it is started again. If the end of the timeline is reached, the `SampleSystem` throws an exception on sample retrieval and playback is automatically paused.

The SceneController forwards the `BikeState` sample to all controllers in the scene implementing the `IBikeVisualizer` interface by way of an event call. The sample is distributed as a reference from which the controllers only read. Additionally, controllers implementing the `IBikePlaybackVisualizer` interface receive event calls whenever the current playback is stopped or started. These are sent by the `PlaybackSystem` and passed on by the `SceneController`.

5.5 SceneController

As outlined in section 5.2, the `SceneController` is the central management point of the application. Its primary function is to establish the different scene configurations by creating and destroying the required controllers. Furthermore, it keeps track of the ones implementing the `IBikeVisualizer` interface to supply them with the newest `BikeState`.

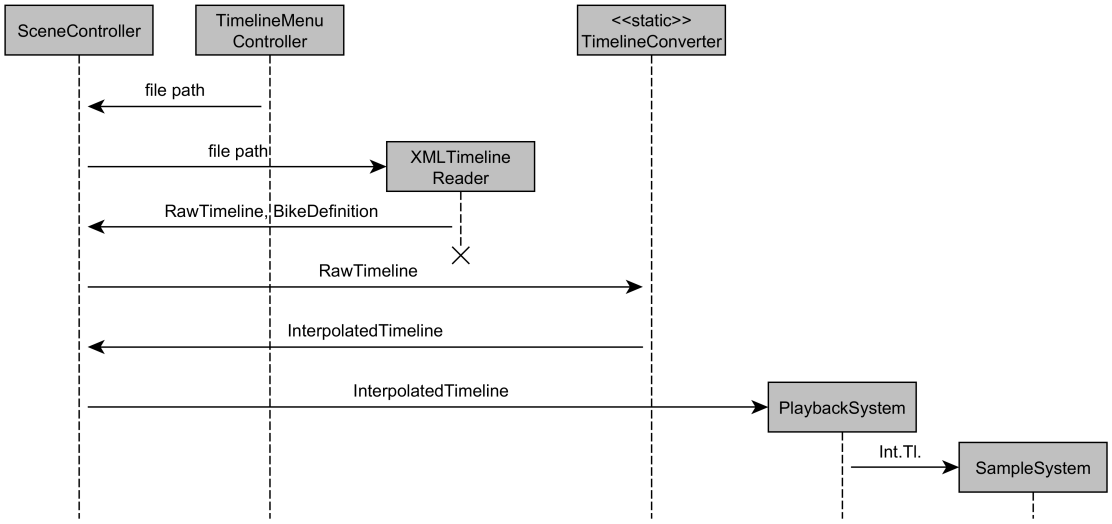


Figure 20: Timeline loading data flow directed by the `SceneController`;

It also handles the data flow for timeline loading, shown in figure 20, by providing a function which invokes the `XMLTimelineReader` from section 4.2. The resulting `RawTimeline` is passed to the `TimelineConverter` (see section 4.3) while saving the retrieved `BikeDefinition` and making it publicly available. The `InterpolatedTimeline` is passed to a new `PlaybackSystem` instance, which in turn sets up a new `SampleSystem` instance. The currently active `PlaybackSystem` is made available to other controllers through a public field. After this process is complete, the application is ready to replay the timeline.

There are two main scenes within the application: The main menu, and the main world. The first consists of the opening screen (`MainMenuController`) and the timeline selection menu (`TimelineMenuController`). The latter includes all bike visualization elements, which are described in detail in section 5.7.1. The settings menu (`SettingsController`) is accessible from both scenes.

5.6 SettingsService

The `SettingsService` is responsible for applying a given settings configuration, and loading and writing it to a file. The following settings are managed by the system:

- Window display mode (fullscreen, borderless windowed, windowed)
- Display resolution (width × height)
- Frames per second limit (value or off)
- UI scale factor
- Pedal offset override (value or -1 for off)
- If the status bars are shown

- If the gears indicator is shown
- If the steering indicator is shown
- If the artificial horizon is shown
- If the compass is shown

The values are set in the settings menu, which is described in section 5.7.3. A given configuration is passed to the service as a `Settings` class instance, which is applied to the application using the following code:

Listing 2: Applying the settings to the application

```
1  //set screen display mode and resolution
2  FullScreenMode fsMode = (FullScreenMode)(settings.FullscreenMode == 2 ? 3
       : settings.FullscreenMode);
3  string[] splitRes = settings.Resolution.Split('x');
4  if (splitRes.Length != 2)
5  throw new ArgumentException("Settings resolution in invalid format. '" +
       settings.Resolution + "'");
6  Screen.SetResolution(Convert.ToInt32(splitRes[0]),
       Convert.ToInt32(splitRes[1]), fsMode);
7
8  //set fps limit
9  QualitySettings.vSyncCount = 0;
10 int target = settings.FPSLimit == "off" ? 0 :
       Convert.ToInt32(settings.FPSLimit);
11 Application.targetFrameRate = target;
12
13 //set UI scale
14 ControllerMaster.instance.gameObject.GetComponent<CanvasScaler>().scaleFactor
       = settings.UIScale;
```

The different UI elements are shown or hidden using the `ControllerMaster`. If the pedal offset override value is not -1, it is set as the current pedal offset for the `SampleSystem`. Otherwise, the one specified by the loaded timeline is used instead.

When a configuration is applied and the menu is closed, it is automatically saved to a file to make the changes permanent. This is done by passing the `Settings` instance to an `XMLSerializer` and writing the output to a file. Similarly, the file is automatically loaded and the configuration applied when the application is started. If no settings file can be found or the settings are incompatible, a default configuration is used instead.

5.7 User Interface and Visual Systems

The user interface follows two main patterns: active and passive colors, and a left-to-right workflow. Interactive elements such as buttons, scroll bars, or input fields have a light gray color, whereas the background is kept in a uniform dark gray. The buttons on interfaces such as the settings or timeline selection menu are set up to have the 'back' or 'cancel' type actions on the left, and the 'confirm' or 'continue' type actions on the right side. An overview of the final result is shown in figure 21.

5.7.1 Bike Visualization Systems

As described in section 5.2, visual systems are separated by function into different controllers. All of them implement either the `IBikeVisualizer` or the derived `IBikePlaybackVisualizer` interface (see section 5.4).

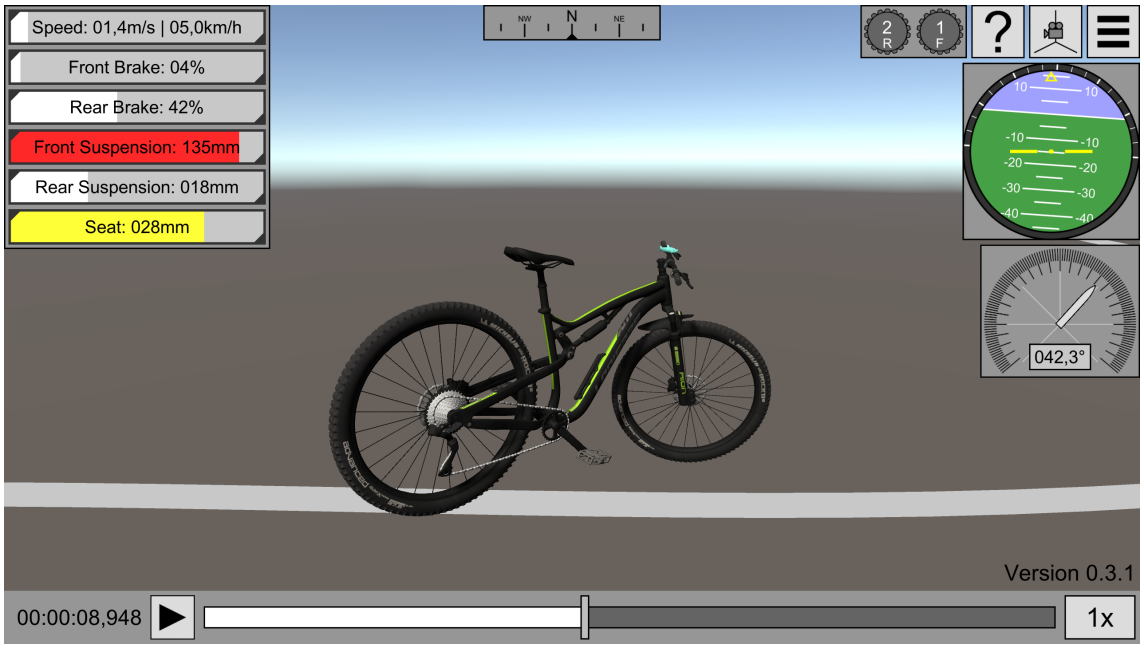


Figure 21: Overview of the main scene user interface.

Bike Animation The main visual feature of the application, the 3 D model, is managed by the `BikeAnimationController`. It updates the `Animator` of the bike object whenever a new `BikeState` is pushed. Using the setup of the `Animator`, as described in section 5.3, the different bike motions are synchronized to the current state. The animations for each bike motion are set to the right position by changing the animation playback position of the corresponding layer. For the wheel rotations, a separate variable controlling the animation speed for the wheel turning is configured. This multiplier is set to correspond to the current revolutions per minute of the wheels, as recorded in the timeline.

Figure 22: A section of bike trail.

Trail Based on the Unity `Line Renderer` [22], the path of the bike is drawn in space by the `TrailController`. The `PlaybackSystem` and `SampleSystem` provide functions for computing a list of positions of the bike over the duration of the timeline. A section of this is shown in figure 22. The resolution for this sampling is set to be a number of

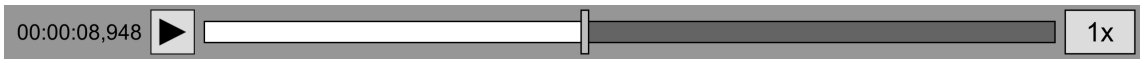


Figure 23: Playback controls with the current time, play/pause button, seek bar, and replay speed button.

samples for every second of recorded ride, with four being the default. These locations are only computed once at the start of the application, and the computation is completely independent from the rest of the `BikeState` sample computation process. The computed list of positions is passed to a `Line Renderer`, which renders the path within the scene.

Playback The `PlaybackController` is responsible for the video-player-like controls within the main scene, as shown in figure 23. The seek bar is created using a Unity `Slider` and the `Event System` to process dragging and clicking. The controller interfaces directly with the current `PlaybackSystem` to pause or start playback, set the current play time, or change the replay speed. The timeline seek bar, play/pause button, and time text are indirectly updated through the `IBikePlaybackVisualizer` event calls to keep them in sync with the rest of the scene.

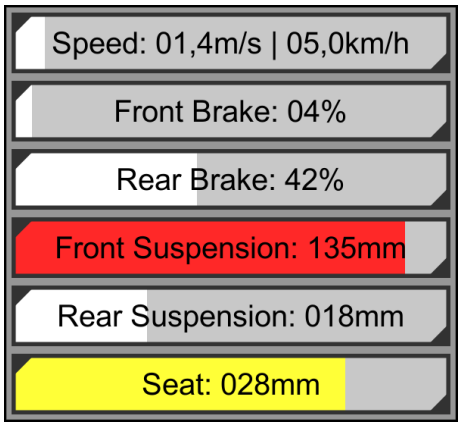


Figure 24: Status bars showing numerical values.

Status Bars Some of the bike properties move between fixed limits. For example, the travel of the front suspension is anywhere between at rest (= 0mm) and full travel (= 150mm). These are shown using bars (see figure 24) that are filled depending on how close to the maximum the value is. The values used for the `BikeState` are converted into the displayed units using the `BikeDefinition` provided by the `SceneController`. Additionally, the infill is colored to be yellow if the value is above 75% of maximum, and red if it's above 90%. The responsible controller is the `BarController`, which updates the bars with each state change. The bars are created by removing the handles from Unity `Sliders` and modifying the color of the infill `Image`.

Artificial Horizon In order to visualize the rotation of the bike in 3 D space, a hint from the aviation industry is taken. Artificial horizons (AHs) (also known as attitude indicators) are commonly used in airplanes to show their orientation relative to the horizon. This is recreated as a UI element using several separate parts, as seen in figure 25: The yellow crosshair and arrow indicating the bike's reference (1), the rotating outer ring indicating

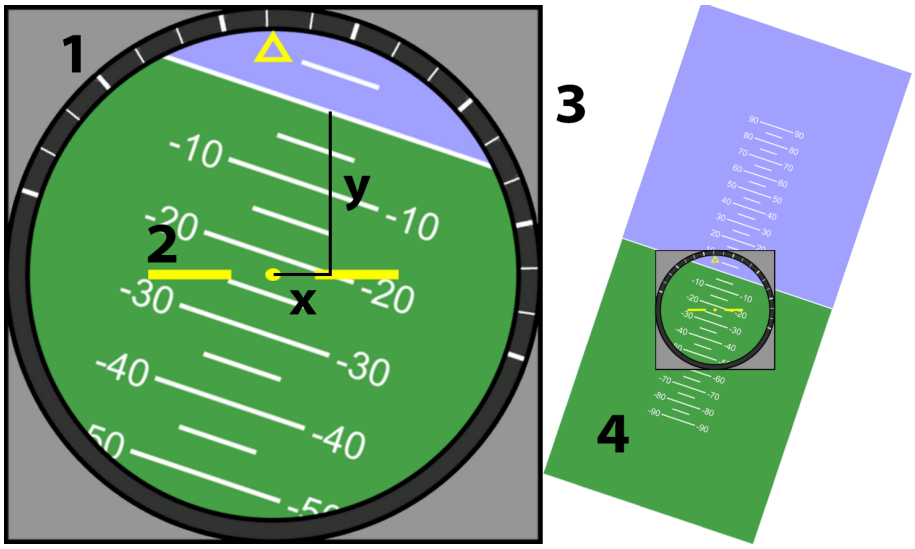


Figure 25: Artificial horizon UI; 1: Rotating outer ring showing side-to-side tilt; 2: Static crosshair; 3: Outside border masking the background overlap; 4: Horizon background which is moved and rotated to match the orientation;

the tilt (2), the front face hiding the overhang of the background through masking (3), and the moving and rotating horizon background with degree demarcations (4). All of this is managed by the `ArtificialHorizonController`. The rotation of the outer ring as well as the horizon background rotation and offset are computed using the code in listing 3.

Listing 3: Computing the rotation and offset of the artificial horizon

```
float tilt = bikeRotation.z % 360f;
//clamp to at most +-90 deg. to either side
if (tilt < -90f) tilt = -90f;
else if (tilt > 90f) tilt = 90f;

//rotate outer ring and horizon background
dialTransform.rotation = Quaternion.Euler(0, 0, tilt);
horizonTransform.rotation = Quaternion.Euler(0, 0, tilt);

//compute front-to-back rotation (=vertical in AH)
float vertRotation = bikeRotation.x % 180f;
//wrap around if past 90 deg. up or down
if (vertRotation > 90) vertRotation -= 180;
else if (vertRotation < -90) vertRotation += 180;

//constant pixel count to move background from 0 to 90 deg.
const float offset90 = -257.2f;

//compute horizon background offsets
Vector2 horizonPosition = new Vector2();
float tiltRotationDeg = tilt * (Mathf.PI / 180);
//vertical offset in pixels
float vertOffset = (offset90 / 90f) * vertRotation;
horizonPosition.x = - Mathf.Sin(tiltRotationDeg) * vertOffset;
horizonPosition.y = Mathf.Cos(tiltRotationDeg) * vertOffset;

//move horizon background
horizonTransform.anchoredPosition = horizonPosition;
```

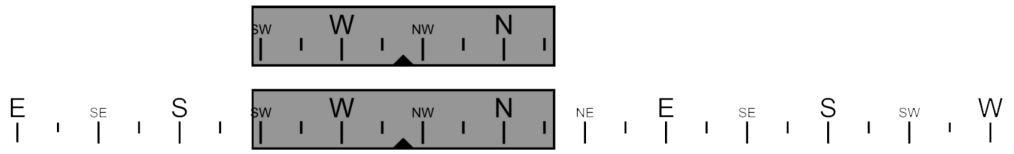
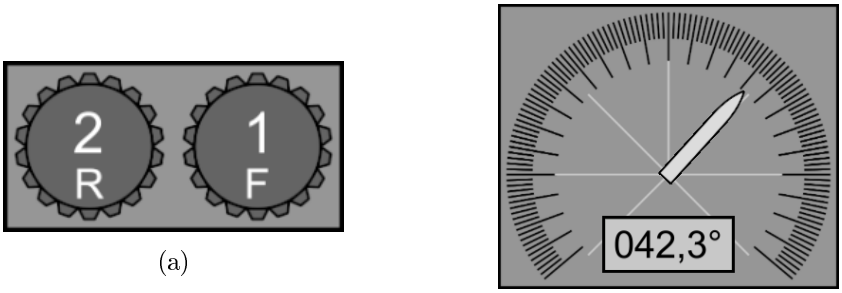


Figure 26: Compass UI; Top: Regular masked display; Bottom: The entire compass image;



(a) (b)

Figure 27: Selected gears (a) and steering rotation (b) indicator interfaces;

Compass The facing direction of the bike is visualized using a horizontally scrolling compass. As shown in figure 26, it consists of a wide image showing the directions, and a panel masking the overlap. The image is moved side-to-side according to the y-axis rotation of the bike. If the rotation moves past south on one side, the picture is shifted to the other side. Because both sides look identical, the illusion of a continuous scrolling compass is given. It is managed by the `CompassController`.

Gears and Steering The currently selected gears are shown using a simple image (shown on in figure 27a) which is updated by the `GearsIndicatorController`. The numbers for the rear and front gears are updated to match the current bike state. Similarly, the steering rotation is shown as a dial with the current angle being displayed underneath (figure 27b). The pointer is rotated to the correct orientation by the `SteeringIndicatorController`.

Help Overlay The help overlay provides short descriptions for the interface elements, as seen in figure 28, and is managed by the `HelpOverlayController`. The text boxes are positioned with the same anchor configuration as the UI elements they describe. This causes them to stay at their relative locations when scaling the interface to different resolutions. The overlay ignores all mouse inputs.

5.7.2 Camera Controls

There are two different camera movement modes available in the main scene: Orbit and free. Both are managed by the `CameraController`.

In orbit mode, the camera rotates around a fixed point on the bike model. The user can click and drag on the screen to rotate the bike. This input is captured using a UI element that covers the entire screen and is placed behind all other UIs. Every frame that the mouse button is held down over the UI the difference in pointer location between the previous and current frame is calculated. This travel is then converted into horizontal and vertical rotation angles using the code in listing 4.

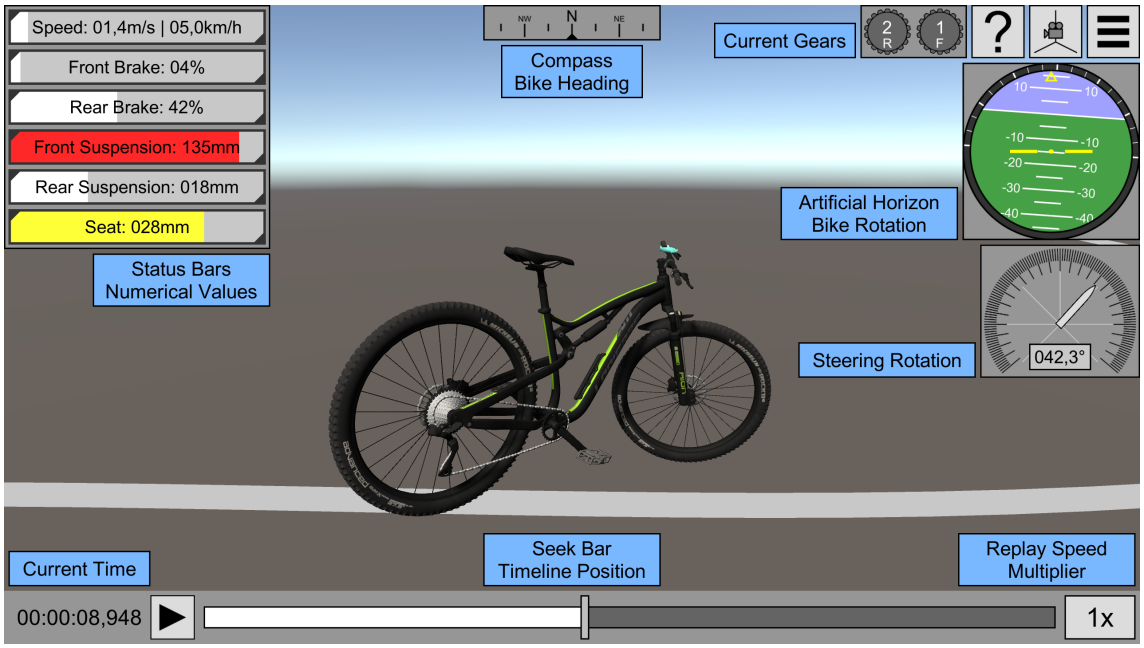


Figure 28: Help overlay with short descriptions for the UI elements;

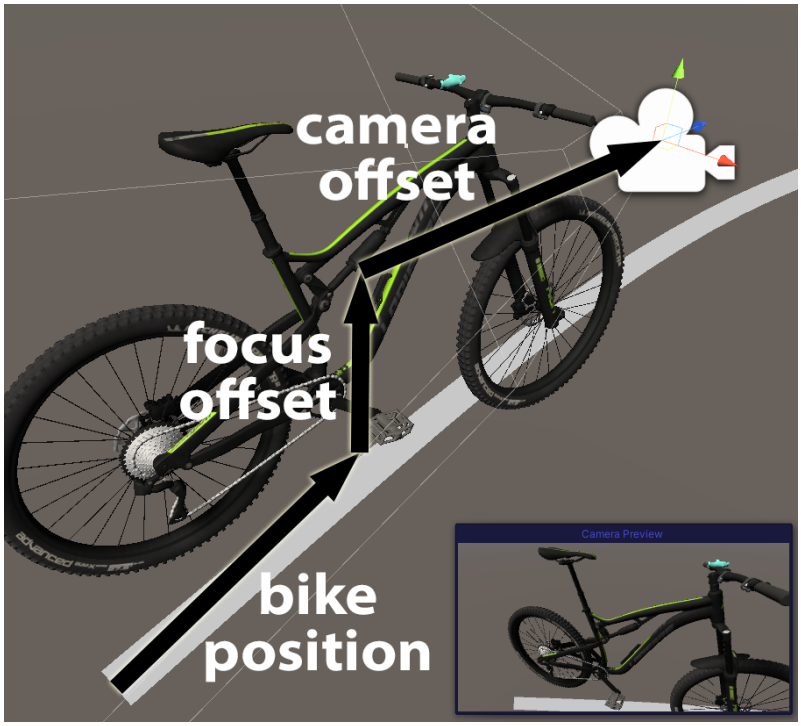


Figure 29: Visualization of the camera position computation.

Listing 4: Computing and updating the camera rotation

```
const float pixelsPerDegree = 10f;

//get mouse movement
Vector3 currentMousePos = Input.mousePosition;
float rot_hor = (currentMousePos.x - lastMousePos.x) / pixelsPerDegree;
float rot_vert = (currentMousePos.y - lastMousePos.y) / pixelsPerDegree;

//add to current camera rotation
cameraAngles.x += rot_vert;
cameraAngles.y += rot_hor;

lastMousePos = currentMousePos;
```

The location of the camera is then computed from the current bike position, the focus point offset, and the current camera offset. This is shown in figure 29. The last one consists of the current distance of the camera from the bike (= zoom level) and the camera rotation angles. All of these are combined into the current camera location using the following code:

Listing 5: Positioning the camera in orbit mode

```
1  //distance from the focus point, modified by scroll wheel input
2  Vector3 baseCameraOffset = new Vector3(-currentCameraDistance, 0, 0);
3
4  //actual offset of the camera from focus point based on distance + rotation
5  Vector3 currentCameraOffset = Quaternion.Euler(new
6      Vector3(0, cameraAngles.y, cameraAngles.x)) * baseCameraOffset;
7
8  //update camera position
9  mainCamera.position = bikeObject.position + currentCameraOffset +
       focusOffset;
10
11 //rotate camera to point at the focus point
12 mainCamera.LookAt(bikeObject.position + focusOffset);
```

The computations of listings 4 and 5 are preformed once every frame as long as the user is dragging to rotate the screen. Changing the distance between the camera and the bike using the scroll wheel only requires the code in listing 5 to be executed. Whenever a new bike state is pushed to the scene only the camera position is updated as in line 9 of listing 5.

When the free camera movement mode is active, the position of the camera is updated independently of the bike. The following code is used to register the input keys for the movement controls using the `InputSystem`:

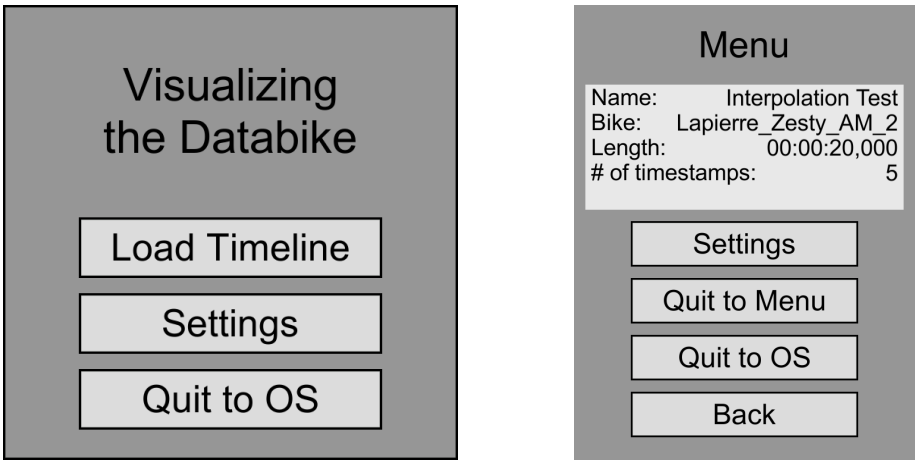
Listing 6: Registering free camera movement controls

```
InputSystem instance = InputSystem.instance;
instance.AddKeyAction("camera_move_forward", KeyCode.W,
    KeyPressType.onIsPressed, () => {
    mainCamera.position +=
        mainCamera.forward * Time.deltaTime * cameraMoveSpeed;
    }
);
instance.AddKeyAction("camera_move_left", KeyCode.A,
    KeyPressType.onIsPressed, () => {
    mainCamera.position +=
        (-mainCamera.right) * Time.deltaTime * cameraMoveSpeed;
    }
);
```

```
14 instance.AddKeyAction("camera_move_right", KeyCode.D,
15     KeyPressType.onIsPressed, () => {
16     mainCamera.position +=
17         mainCamera.right * Time.deltaTime * cameraMoveSpeed;
18     }
19 );
20 instance.AddKeyAction("camera_move_back", KeyCode.S,
21     KeyPressType.onIsPressed, () => {
22     mainCamera.position +=
23         (-mainCamera.forward) * Time.deltaTime * cameraMoveSpeed;
24     }
25 );
```

Note that the position offsets are dependent on a set camera movement speed as well as the time since the last frame update (= Time.deltaTime). The rotation of the camera is still computed using the code from listing 4, but is applied directly to the camera object instead of an offset vector.

5.7.3 User Control Interfaces



(a) (b)

Figure 30: Main menu (a) and escape menu (b) interfaces;

Main Menu When starting the application, the first UI to be displayed is the main menu, managed by the `MainMenuController`. It provides buttons for opening either the settings or timeline menu, and exiting the application, as shown in figure 30a.

Escape Menu The escape menu provides information on the currently loaded timeline as well as buttons for navigation, and is managed by the `EscapeMenuController`. It is shown in figure 30b. Upon opening, playback is automatically paused.

Timeline Menu All timeline files from the "Timelines" folder are listed by the `TimelineMenuController`, as shown in figure 31. The list entries are created and positioned vertically, extending the underlying panel to fit. Entries extending past the end of the list are masked by the surrounding panel, and the attached Unity `Scroll Rect` allows the list to be scrolled vertically. In order to sort the entries by the selected option, a reference to each entry object and content is kept in an internal struct. Once an entry is

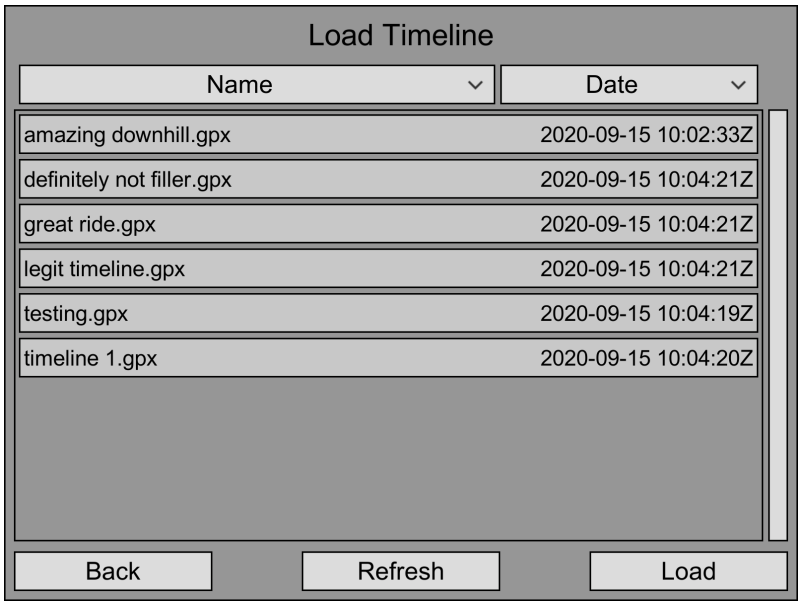


Figure 31: Timeline selection interface.

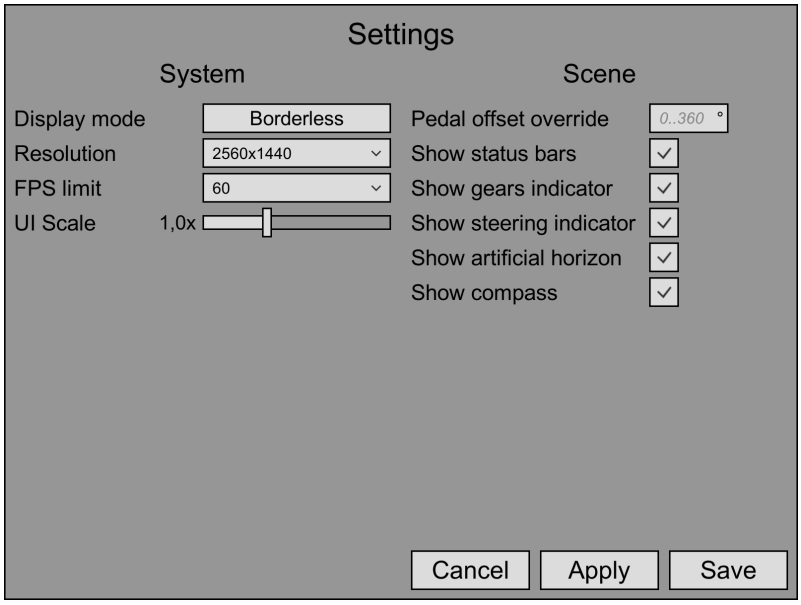


Figure 32: Settings menu.

selected to be loaded, the `SceneController` loads the timeline. If the loading is successful, the scene is switched, otherwise an error message is displayed.

Settings Menu The `SettingsMenuController` is responsible for displaying and reading application settings from the settings menu, as shown in figure 32. The drop down menus for screen resolution and FPS limit are populated upon interface creation using data provided by the `SettingsService`. The settings are read from the interface and passed on as a `Settings` data structure to the `SettingsService`, which then applies them to the application, as described in section 5.6.

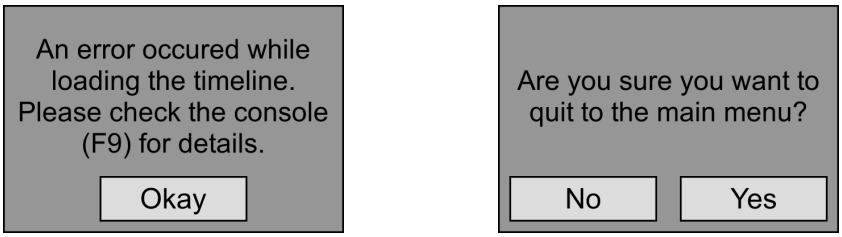


Figure 33: Dialogue windows with either one or two options.

Dialogues The `UniversalDialogueController` provides an easy way to display a dialogue window containing a message and either one or two options to select, as shown in figure 33. The message, number and text of the button(s), as well as the action to be executed when an option is selected can be specified upon creation.

6 Results and Discussion

The resulting application allows the user to load and replay a recorded bike ride. Thanks to the interpolation of the recorded values, playback can be slowed down to review specific sections in detail. The state of the bike is reasonably accurately represented by the animated 3 D model of the Databike. Additionally, most properties are visualized using overlay UI elements.

All user interface elements follow a common theme and are designed to be intuitive to use. Where possible, interface design elements known from other applications are used, such as the seek bar from video players. Interactive elements are recognizable by a light gray background.

The system is developed with expansion in mind: The addition of other bikes is made possible through the `BikeDefinition` file and prefab system. Different timeline formats require only the `TimelineReader` used to parse the file to be switched out. The settings menu is sized to allow for additional options to be added in the future. And, last but not least, additional visualization elements only require the implementation of their respective controller and slight changes to the scene configuration setup.

One potential problem of the current project is the implementation of the `SampleSystem` (see section 4.4). The computation of new samples is performed synchronously in the `FixedUpdate` call in which no more samples are available. This has the potential to freeze up the physics system if it takes too much time to complete. This has not been an issue in testing, however none was conducted on mobile devices with potentially inferior single-threaded performance. One way to ameliorate this problem is to reduce the number of samples to pre-compute, since this spaces the computation out over more, independent calls. A solution would be to outsource the sample computation to another thread which constantly supplies new samples to the system.

In the future the project might be expanded with terrain visualization using video recordings. This would allow the user to see why certain movements were made and put the bike in better context with its surroundings. Another expansion would be to integrate a map view into the main scene to show the current location of the bike. Beyond that, obvious expansions such as supporting more bikes and timeline file formats are easily possible.

References

[1] Blender Foundation. Blender Manual for Armatures. https://docs.blender.org/manual/en/latest/animation/armatures/introduction.html, 2020. [accessed 2020-05-19].

[2] Blender Foundation. Blender Manual for Bone Constraints. https://docs.blender.org/manual/en/latest/animation/armatures/posing/bone_constraints/introduction.html, 2020. [accessed 2020-05-19].

[3] Blender Foundation. Blender Manual for Curves and Surfaces. https://docs.blender.org/manual/en/latest/modeling/curves/introduction.html, 2020. [accessed 2020-05-16].

[4] Blender Foundation. Blender Manual for Inverse Kinematics. https://docs.blender.org/manual/en/latest/animation/armatures/posing/bone_constraints/inverse_kinematics/introduction.html, 2020. [accessed 2020-04-27].

[5] Blender Foundation. Blender Manual for Spline Types. https://docs.blender.org/manual/en/latest/modeling/curves/structure.html#splines, 2020. [accessed 2020-05-16].

[6] Blender Foundation. Blender Manual for the Array Modifier. https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/array.html, 2020. [accessed 2020-05-16].

[7] Blender Foundation. Blender Manual for the Curve Modifier. https://docs.blender.org/manual/en/latest/modeling/modifiers/deform/curve.html, 2020. [accessed 2020-05-16].

[8] Blender Foundation. Blender Manual for the Mirror Modifier. https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/mirror.html, 2020. [accessed 2020-05-16].

[9] Blender Foundation. Blender Manual for the Subdivision Surface Modifier. https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/subdivision_surface.html, 2020. [accessed 2020-05-16].

[10] Blender Foundation. Blender Manual on Keyframes. https://docs.blender.org/manual/en/latest/animation/keyframes/introduction.html, 2020. [accessed 2020-05-22].

[11] Blender Foundation. Blender website. https://www.blender.org, 2020. [accessed 2020-02-13].

[12] Dan Foster. Website GPX: the GPS Exchange Format. https://www.topografix.com/gpx.asp, 2020. [accessed 2020-08-13].

[13] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350 – 355, 1978.

[14] Garmin. Garmin gpx Extensions schema. https://www8.garmin.com/xmlschemas/GpxExtensions/v3/GpxExtensionsv3.xsd, 2020. [accessed 2020-08-13].

[15] Lapierre SAS. Zesty AM 227 website. https://www.bikes-lapierre.de/gamme/2018/mtb/all-mountain/zesty-am-227, 2020. [accessed 2020-02-29].

[16] Lucas Metney / Techcrunch. With new realities to build, Unity positioned to become tech giant. https://techcrunch.com/2017/05/25/with-new-realities-to-build-unity-positioned-to-become-tech-giant, 2017. [accessed 2020-09-20].

[17] Ryan Seghers. C# Cubic Spline Interpolation. https://www.codeproject.com/Articles/560163/Csharp-Cubic-Spline-Interpolation, 2016. [accessed 2020-08-14].

[18] Samuel Picton Drake. Converting GPS Coordinates ($\phi\lambda h$) to Navigation Coordinates (ENU). *DSTO Electronics and Surveillance Research Laboratory*, DSTO-TN-0432, 2002.

[19] Stephen Edelstein / digitaltrends. How gaming company Unity is driving automakers toward virtual reality. https://www.digitaltrends.com/cars/unity-automotive-virtual-reality-and-hmi/, 2019. [accessed 2020-09-20].

[20] Unity Technologies. Unity Manual on GameObjects. https://docs.unity3d.com/Manual/class-GameObject.html, 2020. [accessed 2020-08-11].

[21] Unity Technologies. Unity Manual on the Hierarchy window. https://docs.unity3d.com/Manual/Hierarchy.html, 2020. [accessed 2020-08-11].

[22] Unity Technologies. Unity Manual on the Line Renderer. https://docs.unity3d.com/Manual/class-LineRenderer.html, 2020. [accessed 2020-08-25].

[23] Unity Technologies. Unity Manual on the MonoBehaviours. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html, 2020. [accessed 2020-08-11].

[24] Unity Technologies. Unity Manual on the Prefabs. https://docs.unity3d.com/Manual/Prefabs.html, 2020. [accessed 2020-08-11].

[25] Unity Technologies. Unity Manual on Time Settings. https://docs.unity3d.com/Manual/class-TimeManager.html, 2020. [accessed 2020-09-01].

[26] Univ.-Prof. Dr. Matthias Harders. Lecture Computer Graphics, Chapter Geometry Representation, 2019. Held at the Leopold Franzens University of Innsbruck.

[27] Univ.-Prof. Dr. Matthias Harders. Lecture Computer Graphics, Chapter Texturing, 2019. Held at the Leopold Franzens University of Innsbruck.

[28] Univ.-Prof. Dr. Matthias Harders. Lecture Computer Graphics, Chapter Visibility, 2019. Held at the Leopold Franzens University of Innsbruck.

[29] Wikipedia. Wikipedia article on Spline Interpolation. https://en.wikipedia.org/wiki/Spline_interpolation, 2020. [accessed 2020-08-14].

[30] Wikipedia. Wikipedia article on the Tridiagonal matrix algorithm or Thomas algorithm. https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm, 2020. [accessed 2020-10-06].

[31] C. Yang, T. Lee, C. Huang, and K. Hsu. Unity 3D production and environmental perception vehicle simulation platform. In *International Conference on Advanced Materials for Science and Engineering (ICAMSE)*, pages 452 – 455, 2016.